Practical Guidelines for Solving Difficult Mixed Integer Linear Programs

Ed Klotz† • Alexandra M. Newman‡
†*IBM, 926 Incline Way, Suite 100, Incline Village, NV 89451*
‡*Division of Economics and Business, Colorado School of Mines, Golden, CO 80401*
klotz@us.ibm.com • newman@mines.edu

Abstract

Even with state-of-the-art hardware and software, mixed integer programs can require hours, or even days, of run time and are not guaranteed to yield an optimal (or near-optimal, or any!) solution. In this paper, we present suggestions for appropriate use of state-of-the-art optimizers and guidelines for careful formulation, both of which can vastly improve performance.

"*Problems worthy of attack prove their worth by hitting back.*"
–Piet Hein, Grooks 1966

"*Everybody has a plan until he gets hit in the mouth.*"
–Mike Tyson

Keywords: mixed integer linear programming, memory use, run time, tight formulations, cuts, heuristics, tutorials

1 Introduction

Operations research practitioners have been formulating and solving integer programs since the 1950s. As computer hardware has improved (Bixby and Rothberg, 2007), practitioners have taken the liberty to formulate increasingly detailed and complex problems, assuming that the corresponding instances can be solved. Indeed, state-of-the-art ptimizers such as CPLEX (IBM, 2012), Gurobi (Gurobi, 2012), MOPS (MOPS, 2012), Mosek (MOSEK, 2012), and Xpress-MP (FICO, 2012) can solve many practical large-scale integer programs effectively. However, even if these "real-world" problem instances are solvable in an acceptable amount of time (seconds, minutes or hours, depending on the application), other instances require days or weeks of solution time. Although not a guarantee of tractability, carefully formulating the model and tuning standard integer programming algorithms often result in significantly faster solve times, in some cases, admitting a feasible or near-optimal solution which could otherwise elude the practitioner.

In this paper, we briefly introduce integer programs and their corresponding commonly used algorithm, show how to assess optimizer performance on such problems through the respective algorithmic output, and demonstrate methods for improving that performance through careful formulation and algorithmic parameter tuning. Specifically, there are many mathematically equivalent ways in which to express a model, and each optimizer has its own set of default algorithmic parameter settings. Choosing from these various model expressions and algorithmic settings can profoundly influence solution time. Although it is theoretically possible to try each combination of parameter settings, in practice, random experimentation would require vast amounts of time and would be unlikely to yield significant improvements. We therefore guide the reader to likely performance-enhancing parameter settings given fixed hardware, e.g., memory limits, and suggest methods for avoiding performance failures *a priori* through careful model formulation. All of the guidelines we present here apply to the model in its entirety. Many relaxation and decomposition methods, e.g., Lagrangian Relaxation, Benders' Decomposition, and Column Generation (Dantzig-Wolfe Decomposition), have successfully been used to make large problems more tractable by partitioning the model into subproblems and solving these iteratively. A description of these methods is beyond the scope of our paper; the practitioner should first consider attempting to improve algorithmic performance or tighten the existing model formulation, as these approaches are typically easier and less time consuming than reformulating the model and applying decomposition methods.

The reader should note that we assume basic familiarity with fundamental mathematics, such as matrix algebra, and with optimization, in particular, with linear programming and the concepts contained in Klotz and Newman (To appear). We expect that the reader has formulated linear integer programs and has a conceptual understanding of how the corresponding problems can be solved. Furthermore, we present an algebraic, rather than a geometric, tutorial, i.e., a tutorial based on the mathematical structure of the problem and corresponding numerical algorithmic output, rather than based on graphical analysis. The interested reader can refer to basic texts such as Rardin (1998) and Winston (2004) for more detailed introductions to mathematical programming, including geometric interpretations.

We have attempted to write this paper to appeal to a diverse audience. Readers with limited mathematical programming experience who infrequently use optimization software and do not wish to learn the details regarding how the underlying algorithms relate to model formulations can still benefit from this paper by learning how to identify sources of slow performance based on optimizer output. This identification will allow them to use the tables in the paper that list potential performance problems and parameter settings that address them. More experienced practitioners who are interested in the way in which the optimizer algorithm relates to the model formulation will gain insight into new techniques for improving model formulations, including those different from the ones discussed in this paper. While intended primarily for practitioners seeking performance enhancements to practical models, theoretical researchers may still benefit. The same guidelines that can help tighten specific practical models can also help in the development of the theory associated with fundamental algorithmic improvements in integer programming, e.g., new cuts and new techniques for preprocessing.

The remainder of the paper is organized as follows: In Section 2, we introduce integer programs, the branch-and-bound algorithm, and its variants. Section 3 provides suggestions for successful algorithm performance. Section 4 presents guidelines for and examples of tight formulations of integer programs that lead to faster solution times. Section 5 concludes the paper with a summary. Section 2, with the exception of the tables, may be omitted without loss of continuity for the practitioner interested only in formulation and algorithmic parameter tuning without detailed descriptions of the algorithms themselves. To illustrate the concepts we present in this paper, we show output logs resulting from having run a commercial optimizer on a standard desktop machine. Unless otherwise noted, this optimizer is CPLEX 12.2.0.2, and the machine possesses four single-core 3.0 gigahertz Xeon chips and 8 gigabytes of memory.

2 Fundamentals

Consider the following system in which C is a set of indices on our variables x such that $x_j,\ j \in C$ are nonnegative, continuous variables, and I is a set of indices on the variables x such that $x_j,\ j \in I$ are nonnegative, integer variables. Correspondingly, c_C and A_C are the objective function and left-hand-side constraint coefficients, respectively, on the nonnegative, continuous variables, and c_I and A_I are the objective function and left-hand-side constraint coefficients, respectively, on the nonnegative, integer variables. For the constraint set, the right-hand-side constants, b, are given as an $m \times 1$ column vector.

$$(P_{MIP}): \quad \min \quad c_C^T x_C + c_I^T x_I$$

$$\text{subject to} \quad A_C x_C + A_I x_I = b$$

$$x_C, x_I \geq 0, \quad x_I \ \text{integer}$$

Three noteworthy special cases of this standard mixed integer program are (i) the case in which x_I is binary, (ii) the case in which c_C, A_C, and x_C do not exist and x_I is general integer, and (iii) the case in which c_C, A_C, and x_C do not exist and x_I is binary. Note that (iii) is a special case of

(i) and (ii). We refer to the first case as a mixed binary program, the second case as a pure integer program, and the third case as a binary program. These cases can benefit from procedures such as probing on binary variables (Savelsbergh, 1994), or even specialized algorithms. For example, binary programs lend themselves to some established techniques in the literature that do not exist if the algorithm is executed on an integer program. These techniques are included in most standard branch-and-bound optimizers; however, some features that are specific to binary-only models, e.g., the additive algorithm of Balas (1965), can be lacking.

Branch-and-bound uses intelligent enumeration to arrive at an optimal solution for a (mixed) integer program or any special case thereof. This involves construction of a search tree. Each node in the tree consists of the original constraints in (P_{MIP}), along with some additional constraints on the bounds of the integer variables, x_I, to induce those variables to assume integer values. Thus, each node is also a mixed integer program (MIP). At each node of the branch-and-bound tree, the algorithm solves a linear programming relaxation of the restricted problem, i.e., the MIP with all its variables relaxed to be continuous.

The *root node* at the top of the tree is (P_{MIP}) with the variables x_I relaxed to assume continuous values. Branch-and-bound begins by solving this problem. If the root node linear program (LP) is infeasible, then the original problem (which is more restricted than its linear programming relaxation) is also infeasible, and the algorithm terminates with no feasible solution. Similarly, if the optimal solution to the root node LP has no integer restricted variables with fractional values, then the solution is optimal for (P_{MIP}) as well. The most likely case is that the algorithm produces an optimal solution for the relaxation with some of the integer-restricted variables assuming fractional values. In this case, such a variable, $x_j = f$, is chosen and *branched on*, i.e., two *subproblems* are created – one with a restriction that $x_j \leq \lfloor f \rfloor$ and the other with a restriction that $x_j \geq \lceil f \rceil$. These subproblems are successively solved, which results in one of the following three outcomes:

Subproblem Solution Outcomes (for a minimization problem)

- (i) **The subproblem is optimal with all variables in I assuming integer values**. In this case, the algorithm can update its best integer feasible solution; this update tightens the upper bound on the optimal objective value. Because the algorithm only seeks a single optimal solution, no additional branches are created from this node; examining additional branches cannot yield a better integer feasible solution. Therefore, the node is *fathomed* or *pruned*.

- (ii) **The subproblem is infeasible**. In this case, no additional branching can restore feasibility. As in (i), the node is fathomed.

- (*iii*) **The subproblem has an optimal solution, but with some of the integer-restricted variables in I assuming fractional values**. There are two cases:

  - ⋆ **a.** The objective function value is dominated by the objective of the best integer feasible solution. In other words, the optimal node LP objective is no better than the previously established upper bound on the optimal objective for (P_{MIP}). In this case, no additional branching can improve the objective function value of the node, and, as in (*i*), the node is fathomed.

  - ⋆ **b.** The objective function value is not dominated by that of the best integer feasible solution. The algorithm then *processes* the node in that it chooses a fractional $x_{j'} = f'; \;\; j' \in I$ to branch on by creating two *child* nodes and their associated subproblems – one with a restriction that $x_{j'} \leq \lfloor f' \rfloor$ and the other with a restriction that $x_{j'} \geq \lceil f' \rceil$. These restrictions are imposed on the subproblem in addition to any others from previous branches in the same chain stemming from the root; each of these child subproblems is subsequently solved. Note that while most implementations of the algorithm choose a single integer variable from which to create two child nodes, the algorithm's convergence only requires that the branching divides the feasible region of the current node in a mutually exclusive manner. Thus, branching on groups of variables or expressions of variables is also possible.

Due to the exponential growth in the size of such a tree, exhaustive enumeration would quickly become hopelessly computationally expensive for MIPs with even dozens of variables. The effectiveness of the branch-and-bound algorithm depends on its ability to prune nodes. Effective pruning relies on the fundamental property that the objective function value of each child node is either the same as or worse than that of the parent node (both for the MIP at the node and the associated LP relaxation). This property holds because every child node consists of the MIP in the parent node plus an additional constraint (typically, the bound constraint on the branching variable).

As the algorithm proceeds, it maintains the incumbent integer feasible solution with the best objective function determined thus far in the search. The algorithm performs updates as given in (*i*) of Subproblem Solution Outcomes. The updated incumbent objective value provides an upper bound on the optimal objective value. A better incumbent increases the number of nodes that can be pruned in case (*iii*), part (*a*) by more easily dominating objective function values elsewhere in the tree.

In addition, the algorithm maintains an updated lower bound on the optimal objective for (P_{MIP}). The objective of the root node LP establishes a lower bound on the optimal objective because its feasible region contains all integer feasible solutions to (P_{MIP}). As the algorithm proceeds, it dynamically updates the lower bound by making use of the property that child node objectives are no better than those of their parent. Because a better integer solution can only be produced by the children of the currently unexplored nodes, this property implies that the optimal objective value for (P_{MIP}) can be no better than the best unexplored node LP objective value. As the algorithm continues to process nodes, the minimum LP objective of the unexplored nodes can dynamically increase, improving the lower bound. When the lower bound meets the upper bound, the algorithm terminates with an optimal solution. Furthermore, once an incumbent has been established, the algorithm uses the difference between the upper bound and lower bound to measure the quality of the solution relative to optimality. Thus, on difficult models with limited computation time available, practitioners can configure the algorithm to stop as soon as it has an integer feasible solution within a specified percentage of optimality. Note that most other approaches to solving integer programs (e.g., tabu search, genetic algorithms) lack any sort of bound, although it may be possible to derive one from the model instance. However, even if it is possible to derive a bound, it is likely to be weak, and it probably remains static. Note that in the case of a maximization problem, the best integer solution provides a lower bound on the objective function value and the objective of the root node LP establishes an upper bound on the optimal objective; the previous discussion holds, but with this reversal in bounds. Unless otherwise noted, our examples are minimization problems, as given by our standard form in (P_{MIP}).

Figure 1 provides a tree used to solve a hypothetical integer program of the form (P_{MIP}) with the branch-and-bound algorithm. Only the relevant subset of solution values is given at each node. The numbers in parentheses outside the nodes denote the order in which the nodes are processed, or examined. The inequalities on the arcs indicate the bound constraint placed on an integer-restricted variable in the original problem that possesses a fractional value in a subproblem.

Node (1) is the root node. Its objective function value provides a lower bound on the minimization problem. Suppose x_1, an integer-restricted variable in the original problem, possesses a fractional value (3.5) at the root node solve. To preclude this fractional value from recurring in any subsequent child node solve, we create two subproblems, one with the restriction that $x_1 \leq 3$, i.e., $x_1 \leq \lfloor 3.5 \rfloor$, and the other with the restriction that $x_1 \geq 4$, i.e., $x_1 \geq \lceil 3.5 \rceil$. This is a mutually exclusive and collectively exhaustive set of outcomes for x_1 (and, hence, the original MIP) given that x_1 is an integer-restricted variable in the original problem.

Node (2) is the child node that results from branching down on variable x_1 at node (1). Among possibly others, x_7 is an integer-restricted variable that assumes a fractional value when this sub-

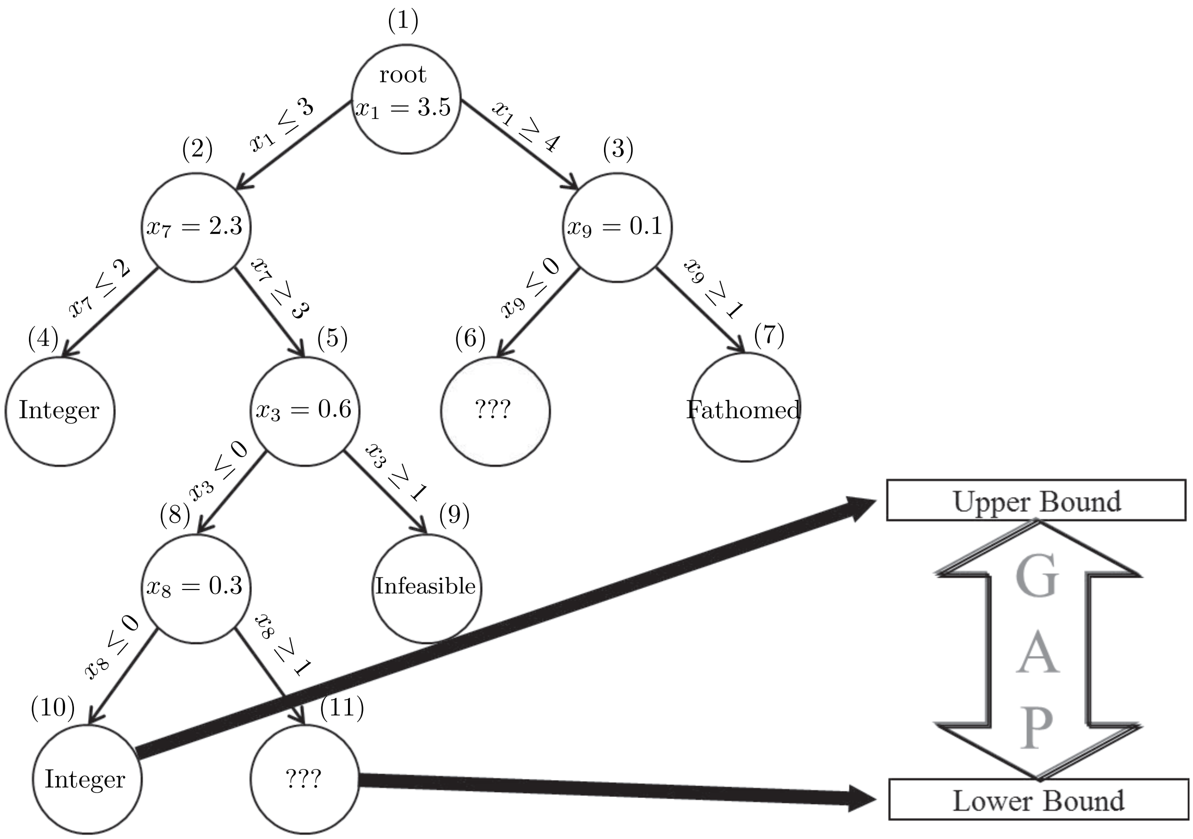


Figure 1: Branch-and-bound algorithm

problem at node (2) is solved; the solve consists of the root node problem and the additional restriction that $x_1 \leq 3$. Because of this fractional value, we create two subproblems emanating from node (2) in the same way in which we create them from node (1). The subproblem solve at node (4), i.e., the solve consisting of the root node subproblem plus the two additional restrictions that $x_1 \leq 3$ and $x_7 \leq 2$, results in an integer solution. At this point, we can update the upper bound. That is, the optimal solution for this problem, an instance of (P_{MIP}), can never yield an objective worse than that of the best feasible solution obtained in the tree.

At any point in the tree, nodes that require additional branching are considered active, or unexplored. Nodes (6) and (11) remain unexplored. Additional processing has led to pruned nodes (4), (7), and (9), either because the subproblem solve was infeasible, e.g., node (9), or because the objective function value was worse than that of node (4), regardless of whether or not the resulting solution was integer. As the algorithm progresses, it establishes an incumbent solution at node (10). Because nodes (6) and (11) remain unexplored, improvement on the current incumbent can only come from the solutions of the subproblems at nodes (6) and (11) or their descendants. The descendants have an objective function value no better than that of either of these two nodes; therefore, the optimal solution objective is bounded by the minimum of the optimal LP objectives of nodes (6) and (11). Without loss of generality, assume node (11) possesses the lesser objective. That objective value then provides a lower bound on the optimal objective for (P_{MIP}). We can continue searching through the tree in this fashion, updating lower and upper bounds, until either the gap is acceptably small, or until all the nodes have been processed.

The previous description of the branch-and-bound algorithm focuses on its fundamental steps. Advances in the last 20 years have extended the algorithm from branch and bound to branch and cut. Branch and cut, the current choice of most integer programming solvers, follows the same steps as branch and bound, but it also can add cuts. Cuts consist of constraints involving linear expressions of one or more variables that are added at the nodes to further improve performance. As long as these cuts do not remove any integer feasible solutions, their addition does not compromise the correctness of the algorithm. If done judiciously, the addition of such cuts can yield significant performance improvements.

3 Guidelines for Successful Algorithm Performance

There are four common reasons that integer programs can require a significant amount of solution time:

- (i) There is lack of node throughput due to troublesome linear programming node solves.
- (ii) There is lack of progress in the best integer solution, i.e., the upper bound.
- (iii) There is lack of progress in the best lower bound.
- (iv) There is insufficient node throughput due to numerical instability in the problem data or excessive memory usage.

By examining the output of the branch-and-bound algorithm, one can often identify the cause(s) of the performance problem. Note that integer programs can exhibit dramatic variations in run time due to seemingly inconsequential changes to a problem instance. Specifically, differences such as reordering matrix rows or columns, or solving a model with the same optimizer, but on a different operating system, only affect the computations at very low-order decimal places. However, because most linear programming problems drawn from practical sources have numerous alternate optimal basic solutions, these slight changes frequently suffice to alter the path taken by the primal or dual simplex method. The fractional variables eligible for branching are basic in the optimal node LP solution. Therefore, alternate optimal bases can result in different branching variable selections. Different branching selections, in turn, can cause significant performance variation if the model formulation or optimizer features are not sufficiently robust to consistently solve the model quickly. This notion of performance variability in integer programs is discussed in more detail in Danna (2008) and Koch et al. (2011). However, regardless of whether an integer program is consistently or only occasionally difficult to solve, the guidelines described in this section can help address the performance problem. We now discuss each potential performance bottleneck and suggest an associated remedy.

3.1 Lack of Node Throughput Due to Troublesome Linear Programming Node Solves

Because processing each node in the branch-and-bound tree requires the solution of a linear program, the choice of a linear programming algorithm can profoundly influence performance. An interior point method may be used for the root node solve; it is less frequently used than the simplex method at the child nodes because it lacks a basis and hence, the ability to start with an initial solution, which is important when processing tens or hundreds of thousands of nodes. However, conducting different runs in which the practitioner invokes the primal or the dual simplex method at the child nodes is a good idea. Consider the following two node logs, the former corresponding to solving the root and child node linear programs with the dual simplex method and the latter with the primal simplex method.

Node Log #1: Node Linear Programs Solved with Dual Simplex

```
        Nodes                                         Cuts/
Node    Left    Objective       IInf    Best Integer  Best Node      ItCnt
0       0       -89.0000        6                     -89.0000       5278
0       0       -89.0000        6                     Fract: 4       12799
0       2       -89.0000        6                     -89.0000       12799
1       1       infeasible                            -89.0000       20767
2       2       -89.0000        5                     -89.0000       27275
3       1       infeasible                            -89.0000       32502
...
8       2       -89.0000        8                     -89.0000       65717
9       1       infeasible                            -89.0000       73714
...
Solution time = 177.33 sec.  Iterations = 73714  Nodes = 10 (1)
```

Node Log #2: Node Linear Programs Solved with Primal Simplex

```
        Nodes                                                Cuts/         ItCnt
Node    Left    Objective       IInf    Best Integer     Best Node
0       0       -89.0000        5                        -89.0000      6603
0       0       -89.0000        5                        Fract: 5      7120
0       2       -89.0000        5                        -89.0000      7120
1       1       infeasible                               -89.0000      9621
2       2       -89.0000        5                        -89.0000      10616
3       1       infeasible                               -89.0000      12963
...
8       2       -89.0000        8                        -89.0000      21522
9       1       infeasible                               -89.0000      23891
...
Solution time = 54.37 sec.  Iterations = 23891  Nodes = 10 (1)
```

The iteration count for the root node solve shown in **Node Log #1** that occurred without any advanced start information indicates 5,278 iterations. Computing the average iteration count across all node LP solves, there are 11 solves (10 nodes, and 1 extra solve for cut generation at node 0) and 73,714 iterations, which were performed in a total of 177 seconds. The summary output in gray indicates in parentheses that one unexplored node remains. So, the average solution time per node is approximately 17 seconds, and the average number of iterations per node is about 6,701. In **Node Log #2**, the solution time is 54 seconds, at which point the algorithm has performed 11 solves, and the iteration count is 23,891. The average number of iterations per node is about 2,172. In **Node Log #1**, the 10 child node LPs require more iterations, 6,844, on average, than the root node LP (which requires 5,278), despite the advanced basis at the child node solves that was absent at the root node solve. Any time this is true, or even when the average node LP iteration count is more than 30-50% of the root node iteration count, an opportunity for improving node LP solve times exists by changing algorithms or algorithmic settings. In **Node Log #2**, the 10 child node LPs require 1,729 iterations, on average, which is much fewer than those required by the root node solve, which requires 6,603 (solving the LP from scratch). Hence, switching from the dual simplex method in **Node Log #1** to the primal simplex method in **Node Log #2** increases throughput, i.e., decreases the average number of iterations required to solve a subproblem in the branch-and-bound tree.

The different linear programming algorithms can also benefit by tuning the appropriate optimizer parameters. See Klotz and Newman (To appear) for a detailed discussion of this topic.

3.2 Lack of Progress in the Best Integer Solution

An integer programming algorithm may struggle to obtain good feasible solutions. **Node Log #3** illustrates a best integer solution found before node 300 of the solve that has not improved by node 7800 of the same solve:

Node Log #3: Lack of Progress in Best Integer Solution

```
       Nodes                                      Cuts/         ItCnt     Gap
Node   Left   Objective   IInf  Best Integer  Best Node
...
300    229     22.6667     40   31.0000       22.0000       4433      29.03%
400    309     cutoff           31.0000       22.3333       5196      27.96%
500    387     26.5000     31   31.0000       23.6667       6164      26.88%
...
7800    5260    28.5000    23   31.0000       25.6667       55739     17.20%
```

Many state-of-the-art optimizers have built-in heuristics to determine initial and improved integer solutions. However, it is always valuable for the practitioner to supply the algorithm with an initial solution, no matter how obvious it may appear to a human. Such a solution may provide a better starting point than what the algorithm can derive on its own, and algorithmic heuristics may perform better in the presence of an initial solution, regardless of the quality of its objective function value. In addition, the faster progress in the cutoff value associated with the best integer solution may enable the optimizer features such as probing to fix additional variables, further improving performance. Common tactics to find such starting solutions include the following:

- Provide an obvious solution based on specific knowledge of the model. For example, models with integer penalty variables may benefit from a starting solution with a significant number (or even all) of the penalty variables set to non-zero values.

- Solve a related, auxiliary problem to obtain a solution (e.g., via the Feasopt method in CPLEX, which looks for feasible solutions by minimizing infeasibilities), provided that the gain from the starting solution exceeds the auxiliary solve time.
- Use the solution from a previous solve for the next solve when solving a sequence of models.

To see the advantages of providing a starting point, compare **Node Log #5** with **Node Log #4**. Log #4 shows that CPLEX with default settings takes about 1589 seconds to find a first feasible solution, with an associated gap of 4.18%. Log #5 illustrates the results obtained by solving a sequence of five faster optimizations (see Lambert et al. (to appear) for details) to obtain a starting solution with a gap of 2.23%. The total computation time to obtain the starting solution is 623 seconds. So, the time to obtain the first solution is faster by providing an initial feasible solution, and if we let the algorithm with the initial solution run for an additional $1589 - 623 = 966$ seconds, the gap for the instance with the initial solution improves to 1.53%.

Node Log #4: No initial practitioner-supplied solution

```
Root relaxation solution time =  131.45 sec.

        Nodes                                             Cuts/
   Node  Left     Objective  IInf  Best Integer     Best Node    ItCnt     Gap

      0     0   1.09590e+07  2424                 1.09590e+07   108111
      0     0   1.09570e+07  2531                     Cuts: 4   108510
      0     0   1.09405e+07  2476                     Cuts: 2   109208
Heuristic still looking.
Heuristic still looking.
Heuristic still looking.
Heuristic still looking.
Heuristic still looking.
      0     2   1.09405e+07  2476                 1.09405e+07   109208
Elapsed real time = 384.09 sec. (tree size =  0.01 MB)
      1     3   1.08913e+07  2488                 1.09405e+07   109673
      2     4   1.09261e+07  2326                 1.09405e+07   109977
...
   1776  1208   1.05645e+07    27                 1.09164e+07   474242
```

```
   1814  1246   1.05588e+07    31                  1.09164e+07   478648
   1847  1277   1.05554e+07   225                  1.09164e+07   484687
*  1880+ 1300                       1.04780e+07   1.09164e+07   491469    4.18%
   1880  1302   1.05474e+07   228   1.04780e+07   1.09164e+07   491469    4.18%
Elapsed real time =  1589.38 sec.  (tree size = 63.86 MB)
```

Node Log #5: An initial solution supplied by the practitioner

```
Root relaxation solution time =   93.92 sec.

        Nodes                                              Cuts/
   Node  Left     Objective  IInf  Best Integer     Best Node    ItCnt     Gap

*     0+    0                       1.07197e+07                 108111     ---
      0     0   1.09590e+07  2424   1.07197e+07   1.09590e+07   108111    2.23%
      0     0   1.09570e+07  2531   1.07197e+07       Cuts: 4   108538    2.21%
...

    485   433   1.09075e+07  2398   1.07197e+07   1.08840e+07   244077    1.53%
    487   434   1.08237e+07  2303   1.07197e+07   1.08840e+07   244350    1.53%
    497   439   1.08637e+07  1638   1.07197e+07   1.08840e+07   245391    1.53%
Elapsed real time =  750.11 sec.  (tree size = 32.61 MB)
    501   443   1.08503e+07  1561   1.07197e+07   1.08840e+07   245895     1.53%
...
Elapsed real time = 984.03 sec. (tree size = 33.00 MB)
   1263   674   1.08590e+07  2574   1.07197e+07   1.08840e+07   314814    1.53%
```

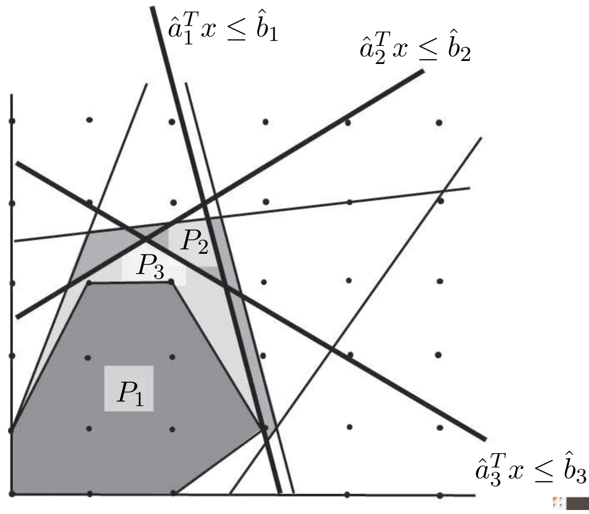
In the absence of a readily identifiable initial solution, various branching strategies can aid in obtaining initial and subsequent solutions. These branching strategies may be based purely on the algebraic structure of the model. For example, by using depth-first search, the branch-and-bound algorithm never defers processing a node until it has been pruned. This strategy helps find integer feasible solutions sooner, although it potentially slows progress in the best bound. (Recall, the best lower bound for a minimization problem is updated once all nodes with relaxation objective value equal to the lower bound have been processed.) In other cases, branching strategies may involve specific aspects of the model. For example, branching up, i.e., processing the subproblem associated with the greater bound as a restriction on its branch, in the presence of many set partitioning constraints ($\sum_i x_i = 1, \quad x_i$ binary) not only fixes the variable on the associated branch in the constraint to 1, but it also fixes all other variables in the constraint to a value of 0 in the children of the current node. By contrast, branching down does not yield the ability to fix any additional variables.

Improvements to the model formulation can also yield better feasible solutions faster. Differentiation in the data, e.g., by adding appropriate discounting factors to cost coefficients in the objective function, helps the algorithm distinguish between dominated and dominating solutions, which expedites the discovery of improving solutions.

3.3 Lack of Progress in the Best Bound

The branch-and-bound depiction in Figure 1 and the corresponding discussion illustrate how the algorithm maintains and updates a lower bound on the objective function value for the minimization integer program. (Note that this would correspond to an upper bound for a maximization problem.) The ability to update the best bound effectively depends on the best objective function value of all active subproblems, i.e., the associated LP objective function value of the nodes that have not been fathomed. If successive subproblems, i.e., subproblems corresponding to nodes lying deeper in the tree, do not possess significantly worse objective function values, the bound does not readily approach the true objective function value of the original integer program. Furthermore, the greater the number of active, i.e., unfathomed, nodes deeper in the tree, the smaller the chance of a tight bound, which always corresponds to the weakest (lowest, for a minimization problem) objective function value of any active node. These objective function values, and the associated bounds they generate, in turn, depend on the strength of the model formulation, i.e., the difference between the polyhedron associated with the LP relaxation of (P_{MIP}) and the polyhedron consisting of the convex hull of all integer feasible solutions to (P_{MIP}). Figure 2 provides an illustration. The region P_1 represents the convex hull of all integer feasible solutions of the MIP, while P_2 represents the feasible region of the LP relaxation. Adding cuts yields the region P_3, which contains all integer solutions of the MIP, but contains only a subset of the fractional solutions feasible for P_2.

Node log #6 exemplifies progress in best integer solution but not in the best bound:



$P_1 := \text{conv}\{x \in \mathbb{Z}^n \,:\, Ax \leq b, x \geq 0\}$

$P_2 := \{x \in \mathbb{R}^n \,:\, Ax \leq b, x \geq 0\}$

$P_3 := P_2 \cap \{x \in \mathbb{R}^n \,:\, \hat{A}x \leq \hat{b}\}$

Cuts must satisfy

1) $\hat{a}_i^T x \leq \hat{b}_i \qquad \forall x \in P_1$ (validity)

2) $\exists\ x \in P_2 : \hat{a}_i^T x > \hat{b}_i$ (separation)

Figure 2: Convex hull

Node Log #6: Progress in Best Integer Solution but not in the Best Bound

		Nodes				Cuts/	ItCnt	Gap
	Node	Left	Objective	IInf	Best Integer	Best Node		
	300	296	2018.0000	27	3780.0000	560.0000	3703	85.19%
*	300+	296		0	2626.0000	560.0000	3703	78.67%
*	393	368		0	2590.0000	560.0000	4405	78.38%
	400	372	560.0000	291	2590.0000	560.0000	4553	78.38%
	500	472	810.0000	175	2590.0000	560.0000	5747	78.38%
...								
*	7740+	5183		0	1710.0000	560.0000	66026	67.25%
	7800	5240	1544.0000	110	1710.0000	560.0000	66279	67.25%
	7900	5325	944.0000	176	1710.0000	560.0000	66801	67.25%
	8000	5424	1468.0000	93	1710.0000	560.0000	67732	67.25%

To strengthen the bound, i.e., to make its value closer to that of the optimal objective function value of the integer program, we can modify the integer program by adding special constraints. These constraints, or *cuts*, do not excise any integer solutions that are feasible in the unmodified integer program. A cut that does not remove any integer solutions is *valid*. However, the cuts remove portions of the feasible region that contain fractional solutions. If the removed area contains the fractional solution resulting from the LP relaxation of the integer program, we say the cut is *useful* (Rardin, 1998), or that the cut *separates* the fractional solution from the resulting LP relaxation feasible region. In this case, the cut improves the bound by increasing the original LP objective. There are various problem structures that lend themselves to different types of cuts. Thus, we have a general sense of cuts that could be useful. However, without the LP relaxation solution, it is difficult to say a priori which cuts are definitely useful.

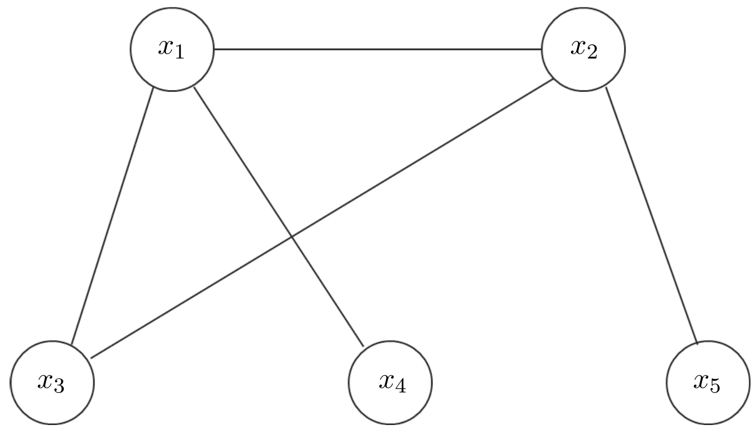


Figure 3: Conflict Graph for numerical example (P^{EX}_{Binary})

Let us consider the following numerical example, in this case, for ease of illustration, a maximization problem:

$$(P^{EX}_{Binary}) \quad \max \quad 3x_1 + 2x_2 + x_3 + 2x_4 + x_5 \tag{1}$$

$$\text{subject to} \quad x_1 + x_2 \leq 1 \tag{2}$$

$$x_1 + x_3 \leq 1 \tag{3}$$

$$x_2 + x_3 \leq 1 \tag{4}$$

$$4x_3 + 3x_4 + 5x_5 \leq 10 \tag{5}$$

$$x_1 + 2x_4 \leq 2 \tag{6}$$

$$3x_2 + 4x_5 \leq 5 \tag{7}$$

$$x_i \ \text{binary} \ \forall i \tag{8}$$

A *cover cut* based on the *knapsack constraint* of (P^{EX}_{Binary}), $4x_3+3x_4+5x_5 \leq 10$, is $x_3+x_4+x_5 \leq 2$. That is, at most two of the three variables can assume a value of 1 while maintaining feasibility of the knapsack constraint (5). Adding this cut is valid since it is satisfied by all integer solutions feasible for the constraint. It also separates the fractional solution $(x_1 = 0,\ x_2 = 0,\ x_3 = 1,\ x_4 = \frac{1}{3},\ x_5 = 1)$ from the LP relaxation feasible region. Now consider the three *packing constraints*, (2) – (4):

$x_1 + x_2 \leq 1$, $x_1 + x_3 \leq 1$, and $x_2 + x_3 \leq 1$. We can construct a *conflict graph* (see Figure 3) for the whole model, with each vertex corresponding to a binary variable and each edge corresponding to a pair of variables, both of which cannot assume a value of 1 in any feasible solution. A clique is a set of vertices such that every two in the set are connected by an edge. At most one variable in a clique can equal 1. Hence, the vertices associated with x_1, x_2 and x_3 form a clique, and we derive the cut: $x_1 + x_2 + x_3 \leq 1$. In addition, constraints (6) and (7) generate the edges $\{1, 4\}$ and $\{2, 5\}$ in the conflict graph, revealing the cuts $x_1 + x_4 \leq 1$ and $x_2 + x_5 \leq 1$. One could interpret these cuts as either clique cuts from the conflict graph, or cover cuts derived directly from constraints (6) and (7). Note that not only does each of these clique cuts separate fractional solutions from the LP relaxation feasible region (as did the cover cut above), but they are also useful in that they remove the LP relaxation solution $(\frac{1}{2},\ \frac{1}{2},\ \frac{1}{2}, \frac{3}{4},\ \frac{7}{8})$ from the feasible region.

The derivations of both clique and cover cuts rely on identifying a linear expression of variables that assumes an integral value in any integer feasible solution, then determining the integer upper (right-hand-side) limit on the expression. In the case of the cover cut for our example (P^{EX}_{Binary}), x_3, x_4 and x_5 form a cover, which establishes that $x_3 + x_4 + x_5 \geq 3$ is infeasible for any integer solution to the model. Therefore, $x_3 + x_4 + x_5 \leq 2$ is valid for any integer feasible solution to (P^{EX}_{Binary}). Similarly, the clique in the conflict graph identifies the integral expression $x_1 + x_2 + x_3$ and establishes that $x_1 + x_2 + x_3 \geq 2$ is infeasible for any integer solution to the model. Therefore, $x_1 + x_2 + x_3 \leq 1$ is valid for any integer feasible solution to (P^{EX}_{Binary}). This cut removes fractional solutions such as $(x_1 = \frac{1}{2},\ x_2 = \frac{1}{2},\ x_3 = \frac{1}{2})$. Making use of fractional infeasibility relative to integer expressions is a useful technique for deriving additional cuts, and is a special case of disjunctive programming (Balas, 1998).

Another mechanism to generate additional cuts includes the examination of the *complementary system*, i.e., one in which a binary variable x_i is substituted with $1 - x_i$. Consider a constraint similar to the knapsack constraint, but with the inequality reversed: $\sum_i a_i x_i \geq b$ (with $a_i, b > 0$). Let $\bar{x}_i = 1 - x_i$. Multiplying the inequality on the knapsack-like constraint by -1 and adding $\sum_i a_i$ to both sides, we obtain: $\sum_i a_i - \sum_i a_i x_i \leq -b + \sum_i a_i$. Substituting the complementary variables yields: $\sum_i a_i \bar{x}_i \leq -b + \sum_i a_i$. Note that when the right hand side is negative, the original constraint is infeasible. Otherwise, this yields a knapsack constraint on $\bar{x}_i$ from which cuts can be derived. Cover cuts involving the $\bar{x}_i$ can then be translated into cuts involving the original x_i variables.

We summarize characteristics of these and other potentially helpful cuts in Table 1. A detailed discussion of each of these cuts is beyond the scope of this paper; see Achterberg (2007) or Wolsey (1998) for more details, as well as extensive additional references. State-of-the-art optimizers tend to implement cuts that are based on general polyhedral theory that applies to all integer programs, or on special structure that occurs on a sufficiently large percentage of practical models. Table 1 can help the practitioner distinguish cuts that a state-of-the-art optimizer is likely to implement from those that are specific to particular types of models, and are less likely to be implemented in a generic optimizer (and, hence, more likely to help performance if the practitioner uses his knowledge to derive them).

Cut name	Mathematical description of cut	Structure of original MILP that generates the cut
Clique†	$\sum_i z_i \leq 1$	Packing constraints
Cover†	$\sum_i z_i \leq b$, b integer	Knapsack constraints
Disjunctive*	Constraint derived from an LP solution	$\sum_i a'_i x_i \geq b'$ or $\sum_i a''_i x_i \geq b''$, x_i continuous or integer
Mixed Integer Rounding*	Use of floors and ceilings of coefficients and integrality of original variables	$a_C x_C + a_I x_I = b$, $x \geq 0$
Generalized Upper Bound†	$\sum_i x_i \leq b$, b integer	Knapsack constraints with precedence or packing
Implied Bound†	$x_i \leq \frac{b}{a_i}$	$\sum_i a_i x_i \leq bz$, $x \geq 0$
Gomory*	Mixed integer rounding applied to a simplex tableau row $\bar{a}$ associated with optimal node LP basis	$\bar{a}_C x_C + \bar{a}_{I/k} x_{I/k} + x_k = \bar{b}$, x_k integer, $x \geq 0$
Zero-half*	$\lambda^T Ax \leq \lfloor \lambda^T b \rfloor$, $\lambda_i \in \{0, 1/2\}$	Constraints containing integer variables and coefficients
Flow Cover†	Linear combination of flow and binary variables involving a single node	Fixed charge network
Flow Path†	Linear combination of flow and binary variables involving a path of nodes	Fixed charge network
Multicommodity flow†	Linear combination of flow and binary variables involving nodes in a network cut	Fixed charge network

Table 1: Different types of cuts and their characteristics, where z is binary unless otherwise noted, and x is continuous; *based on general polyhedral theory; †based on specific, commonly occurring problem structure

Adding cuts does not always help branch-and-bound performance. While it can remove integer infeasibilities, it also results in more constraints in each node LP. More constraints can increase the time required to solve these linear programs. Without a commensurate speed-up in solution time associated with processing fewer nodes, cuts may not be worth adding. Some optimizers have internal logic to automatically assess the trade-offs between adding cuts and node LP solve time. However, if the optimizer lacks such logic or fails to make a good decision, the practitioner may need to look at the branch-and-bound output in order to assess the relative increase in performance due to fewer examined nodes and the potential decrease in the rate at which the algorithm processes the nodes. In other cases, the computational effort required to derive the cuts needed to effectively solve the model may exceed the performance benefit they provide. Similar to node LP solve time and node throughput, a proper comparison of the reduction in solution time the cuts provide with the time spent calculating them may be necessary. (See Achterberg (2007).)

Most optimizers offer parameter settings that can improve progress of the best node, either by strengthening the formulation or by enabling more node pruning. Features that are commonly available include:

- *(i)* **Best Bound node selection** By selecting the node with the minimal relaxation objective value, the algorithm updates the best node value faster. However, by considering node LP objective values while ignoring the number of integer infeasibilities, best bound node selection may cause the optimizer to find fewer integer feasible solutions. Therefore, best bound node selection is most likely to help performance on models in which the optimizer finds integer feasible solutions easily, but has trouble making sufficient progress in the best node.

- *(ii)* **Strong branching** By running a modest number of dual simplex iterations on multiple branching variable candidates at each node, the algorithm can exploit any infeasible branches to tighten additional variable bounds, resulting in a stronger formulation of the MIP at the node in question, and faster pruning of its descendants. Strong branching increases the computation at each node, so the performance improvement from the additional node pruning must compensate for the diminished rate of node throughput to make this a reasonable feature to employ.

- *(iii)* **Probing** By fixing a binary variable to a value of 0 or 1 and propagating this bound change to other variables through the intersecting constraints, the optimizer can often identify binary variables that can only assume one value in any feasible solution. For example, if fixing a binary variable to 0 establishes that (P_{MIP}) is infeasible, then the variable must be 1 in any integer feasible solution. Probing computation time primarily occurs as a preprocessing step before starting the branch-and-bound algorithm. Identifying binary variables to fix can tighten the formulation and improve node throughput by reducing the size of the problem. However, it can be computationally expensive, so the practitioner must compare the time spent performing the initial probing computations with the subsequent performance gains.

- *(iv)* **More aggressive levels of cut generation** Generating more cuts can further tighten the formulation. However, the practitioner must properly assess the trade-off between the tighter formulation and the potentially slower rate of node processing due to the additional constraints in the node LPs.

If alternate parameter settings are insufficient to yield progress in the best node, the following guidelines, while requiring more work, can help address this performance problem:

- (i) **Careful model formulation** It is sometimes possible to use alternate variable definitions. For example, in Bertsimas and Stock Patterson (1998), the authors use variables to denote whether an aircraft (flight) has arrived at a sector in the airspace *by* time period t, and postulate that the variables represented in this manner "define connectivity constraints that are facets of the convex hull of solutions," which greatly improves the tractability of their model. Similarly, in a model designed to determine a net present value-maximizing schedule for extracting three-dimensional notional blocks of material in an open pit mine, we can define $x_{bt} = 1$ if block b is extracted *by* time period t, 0 otherwise, as opposed to the more intuitive $\hat{x}_{bt} = 1$ if block b is extracted *at* time period t, 0 otherwise (Lambert et al., to appear). The definitions in these two references result in models with significant differences in performance, as illustrated theoretically and empirically.

- (ii) **Careful use of elastic variables, i.e., variables that relax a constraint by allowing for violations (which are then penalized in the objective)** Adding elastic variables can result in MIPs that remove the infeasibilities on integer expressions essential to standard cut generation. This leads to a weaker model formulation in which most cut generation mechanisms are disabled. If the use of elastic variables is necessary, consider first minimizing the sum of the elastic variables, then optimizing the original objective while constraining the elastic variable values to their minimized values.

3.4 Data and Memory Problems

Because the optimizer solves linear programs at each node of the branch-and-bound tree, the practitioner must be careful to avoid the numerical performance issues described in Section 3 of Klotz and Newman (To appear). Specifically, it is important to avoid large differences in orders of magnitude in data to preclude the introduction of unnecessary round-off error. Such differences of input values create round-off error in floating point calculations which makes it difficult for the algorithm to distinguish between this error and a legitimate value. If the algorithm makes the wrong distinction, it arrives at an incorrect solution. Integer programs may contain the construct "if $z = 0$, then $x = 0$. Otherwise, x can be arbitrarily large." Arbitrarily large values of x can be carelessly modeled with a numerical value designed to represent infinity (often referred to as "big M" in the literature). In reality, the value for this variable can be limited by other constraints in the problem; if so, we reduce its value, as in the following:

$$x - 100000000000z \leq 0 \tag{9}$$

$$0 \leq x \leq 5000;\ z \text{ binary} \tag{10}$$

In this case, we should use a coefficient of 5000 on z, which allows us to eliminate the explicit upper bound on x as well. In addition to improving the scaling of the constraint, this change to the numerical value enables the optimizer to better identify legitimate solutions to the conditions being modeled. For example, the unmodified constraint accepts values of $z = 10^{-8}$ and $x = 1000$ as an integer feasible solution. Most optimizers use an integrality tolerance and, by default, accept an integrality violation of this order of magnitude. Therefore, the big M coefficient on the original constraint enables the optimizer to accept a solution that, while feasible in a finite precision computing environment, does not satisfy the intended meaning of the constraint. See Camm et al. (1990) for further discussion.

Branch-and-bound can be generalized to other logic, which is important because it removes the urge to use these numerically problematic "big M's" by allowing, for example, direct branching on an indicator constraint. The indicator formulation of (9) is $z = 0 \Rightarrow x \leq 0$. An indicator infeasibility that requires branching occurs when a node relaxation solution has $z = 0$ but $x > 0$. The indicator branches would be: $x \leq 0$ and $z = 1$. By contrast, large values in (9) or elsewhere in the model (whether truly infinite or some big M approximation) can result in a wide range of coefficients that can easily lead to numerical problems. So, using indicators eliminates these potentially large values from the matrix coefficients used to approximate an infinite value. For the case in which the large values impose meaningful limits in the model, the indicator formulation moves the coefficients from the matrix into the variable bounds, which improves the numerical characteristics of the model.

Indicator constraints also support more general conditions, e.g., $z = 0 \Rightarrow a^T x \leq b$. In this case, the indicator branches would be $a^T x \leq b$ and $z = 1$. However, relaxations of indicator constraints remove the constraint completely and can therefore be potentially weaker than their less numerically stable big M counterpart. As of this writing, recent improvements in indicator preprocessing in CPLEX have helped address this drawback.

Integer programs require at least as much memory as their linear programming equivalents. Running out of memory is therefore as frequent, if not more frequent, a problem when trying to solve integer programs, as opposed to linear programs. The same suggestions as those that appear in Subsection 3.3 of Klotz and Newman (To appear) apply.

Table 2 provides suggestions for the branch-and-bound settings to use under the circumstances mentioned in this section.

Characteristic	Recognition	Suggested tactic(s)
• Troublesome LPs	• Large iteration counts per node, especially regarding root node solve	• Switch algorithms between primal and dual simplex; if advanced starts do not help simplex, consider barrier method
• Lack of progress in best integer	• Little or no change in best integer solution in log after hundreds of nodes	• Use best estimate or depth-first search • Apply heuristics more frequently • Supply an initial solution • Apply discount factors in the objective • Branch up or down to resolve integer infeasibilities
• Lack of progress in best node	• Little or no change in best node in log after hundreds of nodes	• Use breadth-first search • Use aggressive probing • Use aggressive algorithmic cut generation • Apply strong branching • Derive cuts *a priori* • Reformulate with different variables
• Data and memory problems	• Slow progress in node solves • Out of memory error	• Avoid large differences in size of data • Reformulate "big M" constraints • Rectify LP problems, e.g., degeneracy • Apply memory emphasis setting • Buy more memory

Table 2: Under various circumstances, different formulations and algorithmic settings have a greater chance of faster solution time on an integer programming problem instance.

4 Tighter Formulations

When optimizer parameter settings (including aggressive application of cuts) fail to yield the desired improvements, the practitioner may obtain additional performance gains by adding cuts more specific to the model. The cuts added by the optimizer typically rely either on general polyhedral theory that applies to all MIPs, or on special structure that appears in a significant percentage of MIPs. In some cases, the cuts needed to improve performance rely on special structure specific to individual MIPs. These less applicable cuts are unlikely to be implemented in any state-of-the-art optimizer. In such cases, the practitioner may need to formulate his own cuts, drawing on specific model knowledge. One can find a staggering amount of theory on cut derivation in integer programming (Grötschel, 2004). While more knowledge of sophisticated cut theory adds to the practitioner's quiver of tactics to improve performance, run time enhancements can be effected with some fairly simple techniques, provided the practitioner uses them in a disciplined, well organized fashion. To that end, this section describes guidelines for identifying cuts that can tighten a formulation of (P_{MIP}) and yield significant performance improvements. These guidelines can help both novice practitioners and those who possess extensive familiarity with the underlying theories of cut generation. See Rebennack et al. (2012) for an example of adding cuts based on specific model characteristics.

Before tightening the formulation, the practitioner must identify elements of the model that make it difficult, specifically, those that contain the constraints and variables from which useful cuts can be derived. The following steps can help in this regard.

Determining How a MIP Can Be Difficult to Solve

- (*i*) **Simplify the model if necessary**. For example, try to identify any constraints or integrality restrictions that are not involved in the slow performance by systematically removing constraints and integrality restrictions and solving the resulting model. Such filtering can be done efficiently by grouping similar constraints and variables and solving model instances with one or more groups omitted. If the model remains difficult to solve after discarding a group of constraints, the practitioner can tighten the formulation without considering those constraints. Or, he can try to reproduce the problem with a smaller instance of the model.

- (*ii*) **Identify the constraints that prevent the objective from improving**. With a minimization problem, this typically means identifying the constraints that force activities to be performed. In other words, practical models involving nonnegative cost minimization inevitably have some constraints that prevent the trivial solution of zero from being viable.

- (*iii*) **Determine how removing integrality restrictions allows the root node relaxation objective to improve**. In weak formulations, the root node relaxation objective tends to be significantly better than the optimal objective of the associated MIP. The variables with fractional solutions in the root node relaxation help identify the constraints and variables that motivate additional cuts. Many models have a wealth of valid cuts that could be added purely by examining the model. But, many of those cuts may actually help little in tightening the formulation. By focusing on how relaxing integrality allows the objective to improve, the practitioner focuses on identifying the cuts that actually tighten the formulation.

Having identified the constraints and variables most likely to generate good cuts, the practitioner faces numerous ways to derive the cuts. While a sophisticated knowledge of the literature provides additional opportunities for tightening formulations, practitioners with limited knowledge of the underlying theory can still effectively tighten many formulations using some fairly simple techniques.

Model Characteristics from which to Derive Cuts

- (*i*) **Linear or logical combinations of constraints** By combining constraints, one can often derive a single constraint in which fractional values can be rounded to produce a tighter cut. The clique cuts previously illustrated with the conflict graph provide an example of how to identify constraints to combine. The conflict graph in that example occurs in a sufficient number of practical MIPs so that many state-of-the-art optimizers use it. But, other MIPs may have different graphs associated with their problem structure that do not occur frequently. Identifying such graphs and implementing the associated cuts can often tighten the formulation and dramatically improve performance.

- (*ii*) **The optimization of one or more related models** By optimizing a related model that requires much less time to solve, the practitioner can often extract useful information to apply to the original model. For example, minimizing a linear expression involving integer variables and integer coefficients can provide a cut on that expression. This frequently helps on models with integer penalty variables.

- (*iii*) **Use of the incumbent solution objective value** Because cuts are often based on infeasibility, models with soft constraints that are always feasible can present unique challenges for deriving cuts. However, while any solution is feasible, the incumbent solution objective value allows the practitioner to derive cuts based on the implicit, dynamic constraint defined by the objective function and the incumbent objective value.

- (*iv*) **Disjunctions** Wolsey (1998) provides a description of deriving cuts from disjunctions, which were first developed by Balas (1998). In general, suppose $X_1 = \left\{x : a^T x \geq b\right\}$ and $X_2 = \left\{x : \hat{a}^T x \geq \hat{b}\right\}$. Let u be the componentwise maximum of a and $\hat{a}$, i.e., $u_j = \max\{a_j, \hat{a}_j\}$. And, let $\bar{u} = \min\left\{b, \hat{b}\right\}$. Then
$$u^T x \geq \bar{u} \tag{11}$$
is valid for $X_1 \cup X_2$, which implies it is also valid for the convex hull of X_1 and X_2. These properties of disjunctions can be used to generate cuts in practice.

- (*v*) **The exploitation of infeasibility** As previously mentioned, cover, clique and other cuts can be viewed as implicitly using infeasibility to identify cuts to tighten a formulation of (P_{MIP}). Generally, for any linear expression involving integer variables with integer coefficients and an integer right hand side b, if $a^T x \leq b$ can be shown to be infeasible, then the constraint $a^T x \geq b + 1$ provides a valid cut.

We now consider a simple example to illustrate the use of disjunctions to derive cuts. Most state-of-the-art optimizers support mixed integer rounding cuts, both on constraints explicitly in the model, and as Gomory cuts based on implicit constraints derived from the simplex tableau rows of the node LP subproblems. So, practitioners typically do not need to apply disjunctions to derive cuts on constraints like the one in the example we describe below. However, we use this simple example to aid in the understanding of the more challenging example we present subsequently. In the first instance, we illustrate the derivation of a mixed integer rounding cut on the constraint:

$$4x_1 + 3x_2 + 5x_3 = 10 \tag{12}$$

$$x_1, x_2, x_3 \geq 0, \text{ integer} \tag{13}$$

Dividing by the coefficient of x_1, we have

$$x_1 + \frac{3}{4}x_2 + \frac{5}{4}x_3 = \frac{5}{2} \tag{14}$$

Now, we separate the left and right hand sides into integer and fractional components, and let $\hat{x}$ represent the integer part of the left hand side:

$$\underbrace{x_1 + x_2 + x_3}_{\hat{x}} - \frac{1}{4}x_2 + \frac{1}{4}x_3 = 2 + \frac{1}{2} = 3 - \frac{1}{2} \tag{15}$$

We examine a disjunction on the integer expression $\hat{x}$. If $\hat{x} \leq 2$, the terms with fractional coefficients on the left hand side of (15) must be greater than or equal to the first fractional term in the right-hand-side expressions. Similarly, the terms with fractional coefficients on the left hand side must be less than or equal to the second fractional term in the right-hand-side expressions if $\hat{x} \geq 3$. Using the nonnegativity of the x variables to simplify the constraints implied by the disjunction, we conclude:

$$\hat{x} \leq 2 \Rightarrow \frac{-1}{4}x_2 + \frac{1}{4}x_3 \geq \frac{1}{2} \Rightarrow x_3 \geq 2 \tag{16}$$

$$\hat{x} \geq 3 \Rightarrow \frac{-1}{4}x_2 + \frac{1}{4}x_3 \leq \frac{-1}{2} \Rightarrow x_2 \geq 2 \tag{17}$$

So, either $x_3 \geq 2$ or $x_2 \geq 2$. We can then use the result of (11) to derive the cut

$$x_2 + x_3 \geq 2 \tag{18}$$

Note that this eliminates the fractional solution $(2, \frac{1}{3}, \frac{1}{5})$, which satisfies the original constraint, (12). Note also that by inspection the only two possible integer solutions to this constraint are $(1, 2, 0)$ and $(0, 0, 2)$. Both satisfy (18), establishing that the cut is valid. (Dividing (12) by the coefficient on x_2 or x_3 instead of x_1 results in a similar mixed integer rounding cut.)

This small example serves to illustrate the derivation of a mixed integer rounding cut on a small constraint; state-of-the-art optimizers such as CPLEX would have been able to identify this cut. However, disjunctions are more general, and can yield performance-improving cuts on models for which the optimizer's cuts do not yield sufficiently good performance. For example, consider the following single-constraint knapsack model. Cornuejols et al. (1997) originally generated this instance. (See Aardal and Lenstra (2004) for additional information on these types of models.) We wish to either find a feasible solution or prove infeasibility for the single-constraint integer program:

$$13429x_1 + 26850x_2 + 26855x_3 + 40280x_4 + 40281x_5 + 53711x_6 + 53714x_7 + 67141x_8 = 45094583$$

$$x_j \geq 0, \text{ integer}, j = 1, \ldots, 8$$

Running CPLEX 12.2.0.2 with default settings results in no conclusion after over 7 hours and 2 billion nodes, as illustrated in **Node Log #7**:

Node Log #7

```
             Nodes                                               Cuts/
   Node       Left      Objective  IInf  Best Integer     Best Node    ItCnt     Gap
...
 2054970910 13066         0.0000     1                      0.0000 25234328
Elapsed real time = 27702.98 sec. (tree size =  2.70 MB, solutions = 0)
 2067491472 14446         0.0000     1                      0.0000 25388082
 2080023238 12892         0.0000     1                      0.0000 25542160
 2092548561 15366         0.0000     1                      0.0000 25696280
...
                          -------
Total (root+branch&cut) = 28302.29 sec.
```

```
MIP - Node limit exceeded, no integer solution.
Current MIP best bound =  0.0000000000e+00 (gap is infinite)
Solution time = 28302.31 sec.  Iterations = 25787898  Nodes = 2100000004 (16642)
```

However, note that all the coefficients in the model are very close to integer multiples of the coefficient of x_1. Therefore, we can separate the left hand side into the part that is an integer multiple of this coefficient, and the much smaller remainder terms:

$$13429\underbrace{(x_1 + 2x_2 + 2x_3 + 3x_4 + 3x_5 + 4x_6 + 4x_7 + 5x_8)}_{\hat{x}} \tag{19}$$

$$-8x_2 - 3x_3 - 7x_4 - 6x_5 - 5x_6 - 2x_7 - 4x_8 \tag{20}$$

$$= 3358 * 13429 + 1 = 3359 * 13429 - 13428 \tag{21}$$

This constraint resembles the one from which we previously derived the mixed integer rounding cut. But, instead of separating the integer and fractional components, we separate the components that are exact multiples of the coefficient of x_1 from the remaining terms. We now perform the disjunction on $\hat{x}$ in an analogous manner, again using the nonnegativity of the variables.

$$\hat{x} \le 3358 \Rightarrow \underbrace{-8x_2 - 3x_3 - 7x_4 - 6x_5 - 5x_6 - 2x_7 - 4x_8}_{\le 0} \ge 1 \tag{22}$$

Thus, if $\hat{x} \le 3358$, the model is infeasible. Therefore, infeasibility implies that $\hat{x} \ge 3359$ is a valid cut. We can derive an additional cut from the other side of the disjunction on $\hat{x}$:

$$\hat{x} \ge 3359 \Rightarrow -8x_2 - 3x_3 - 7x_4 - 6x_5 - 5x_6 - 2x_7 - 4x_8 \le -13428 \tag{23}$$

This analysis shows that constraints (24) (using the infeasibility argument above) and (25) (multiplying (23) through by -1) are globally valid cuts.

$$x_1 + 2x_2 + 2x_3 + 3x_4 + 3x_5 + 4x_6 + 4x_7 + 5x_8 \ge 3359 \tag{24}$$

$$8x_2 + 3x_3 + 7x_4 + 6x_5 + 5x_6 + 2x_7 + 4x_8 \ge 13428 \tag{25}$$

Adding these cuts enables CPLEX 12.2.0.2 to easily identify that the model is infeasible (see **Node**

Log #8). Summarizing this example, concepts (*iv*) and (*v*), the use of disjunctions and exploiting infeasibility, helped generate cuts that turned a challenging MIP into one that was easily solved.

Node Log #8

```
      Nodes                                              Cuts/
   Node  Left     Objective  IInf  Best Integer     Best Node    ItCnt     Gap

      0     0        0.0000     1                      0.0000        1
      0     0        0.0000     2                  MIRcuts: 1        3
      0     0        0.0000     2                  MIRcuts: 1        5
      0     0        cutoff                                          5
Elapsed real time =   0.23 sec. (tree size =  0.00 MB, solutions = 0)
Mixed integer rounding cuts applied:  1
...
MIP - Integer infeasible.
Current MIP best bound is infinite.
Solution time =    0.46 sec.  Iterations = 5  Nodes = 0
```

The second practical example we consider is a rather large maximization problem, and illustrates concepts (*ii*) and (*v*): the optimization of one or more related models and the exploitation of infeasibility, respectively. The example involves a collection of n objects with some measure of distance between them. The model selects $k < n$ of the objects in a way that maximizes the sum of the distances between the selected object, i.e., the k most diverse objects are selected. The most direct model formulation involves binary variables and a quadratic objective. Let $d_{ij} \geq 0$ be the known distance between object i and object j, and let x_i be a binary variable that is 1 if object i is selected, and 0 otherwise. The formulation follows:

$$(MIQP) \quad \max \sum_{i=1}^{n} \sum_{j=i+1}^{n} d_{ij} x_i x_j$$

$$\text{subject to} \quad \sum_{j=1}^{n} x_j \leq k$$

$$x_j \text{ binary}$$

Because this article focuses on linear and linear-integer models, we consider an equivalent linear formulation that recognizes that the product of binary variables is itself a binary variable (Watters, 1967). We replace each product of binaries $x_i x_j$ in $(MIQP)$ with a binary variable z_{ij}, and add constraints to express the relationship between x and z in a mixed integer linear program (MILP):

$$(MILP) \quad \max \sum_{j=1}^{n} \sum_{\substack{i=1 \\ i<j}}^{n} d_{ij} z_{ij} \tag{26}$$

$$\text{subject to} \quad \sum_{j=1}^{n} x_j \leq k \tag{27}$$

$$z_{ij} \leq x_i \quad \forall\, i,j \tag{28}$$

$$z_{ij} \leq x_j \quad \forall\, i,j \tag{29}$$

$$x_i + x_j \leq 1 + z_{ij} \quad \forall\, i,j \tag{30}$$

$$x_j, z_{ij} \text{ binary} \quad \forall\, i,j \tag{31}$$

The constraints (28), (29) and (30) exist for indices (i,j), $i<j$ because the selection of both i and j is equivalent to the selection of both j and i. Hence, the model only defines z_{ij} variables with $i<j$. Note that if x_i or $x_j = 0$, then constraints (28) and (29) force z_{ij} to 0, while (30) imposes no restriction on z_{ij}. Similarly, if both x_i and $x_j = 1$, (28) and (29) impose no restriction on z_{ij}, while (30) forces z_{ij} to 1. So, regardless of the values of x_i and x_j, $z_{ij} = x_i x_j$, and we can replace occurrences of $x_i x_j$ with z_{ij} to obtain the linearized reformulation above.

This linearized model instance with $n = 60$ and $k = 24$ possesses 1830 binary variables, and 5311 constraints. Due to the large branch-and-bound tree resulting from this instance, we set CPLEX's file parameter to instruct CPLEX to efficiently swap the memory associated with the branch-and-bound tree to disk. This enables the run to proceed further than with default settings in which CPLEX stores the tree in physical memory. All other parameter settings remain at defaults, so CPLEX makes use of all four available processors. CPLEX runs for just over four hours (see **Node Log #9**), terminating when the size of the swap file for the branch-and-bound tree exceeds memory limits, i.e., at the point at which CPLEX has processed over 4 million nodes and the solution has an objective value of 3483.0000, proven to be within 51.32% of optimal. This level of performance indicates significant potential for improvement. Although we do not provide the output here, the original MIQP formulation in $(MIQP)$ performs even worse.

Node Log #9

```
        Nodes                                              Cuts/
   Node  Left     Objective  IInf  Best Integer     Best Node    ItCnt     Gap

*     0+    0                             0.0000                  2247     ---
      0     0     7640.4000  1830         0.0000     7640.4000     2247     ---
*     0+    0                            19.0000     7640.4000     2247     ---

...

*     0+    0                          3185.0000     7445.4286     2286  133.77%
      0     2     7628.5333  1829      3185.0000     7445.4286     2286  133.77%
Elapsed real time =   4.09 sec. (tree size =  0.01 MB, solutions = 8)
     35    37     6579.2308  1378      3185.0000     7445.4286     6615  133.77%
...
 4332613 3675298     4936.6750  1099     3483.0000     5270.8377 1.78e+08   51.33%
 4341075 3682375     3889.4643   714     3483.0000     5270.4545 1.79e+08   51.32%

...
CPLEX Error  1803: Failure on temporary file write.

Solution pool: 25 solutions saved.

MIP - Error termination, no tree:  Objective =  3.4830000000e+03
Current MIP best bound =  5.2704102564e+03 (gap = 1787.41, 51.32%)
Solution time = 15031.18 sec.  Iterations = 178699476  Nodes = 4342299 (3682262)
```

Experimentation with non-default parameter settings as described in Section 3 yields modest performance improvements, but does not come close to enabling CPLEX to find an optimal solution to the model.

We carefully examine a smaller model instance with $n = 3$ and $k = 2$ to assess how removing integrality restrictions yields an artificially high objective function value:

$$
\begin{aligned}
\max \quad & 3z_{12} + 4z_{13} + 5z_{23} \\
\text{subject to} \quad & x_1 + x_2 + x_3 \leq 2 \\
& z_{12} - x_1 \leq 0 \\
& z_{12} - x_2 \leq 0 \\
& x_1 + x_2 \leq 1 + z_{12} \\
& z_{13} - x_1 \leq 0 \\
& z_{13} - x_3 \leq 0 \\
& x_1 + x_3 \leq 1 + z_{13} \\
& z_{23} - x_2 \leq 0 \\
& z_{23} - x_3 \leq 0 \\
& x_2 + x_3 \leq 1 + z_{23} \\
& x_1, x_2, x_3, z_{12}, z_{13}, z_{23} \text{ binary}
\end{aligned}
$$

The optimal solution to this MILP consists of setting $z_{23} = x_2 = x_3 = 1$, yielding an objective value of 5. By contrast, relaxing integrality enables a fractional solution consisting of setting all x and z variables to 2/3, yielding a much better objective value of 8. Note that the difference between the MILP and its relaxation occurs when the z_{ij} variables assume values strictly less than 1. When any $z_{ij} = 1$, the corresponding x_i and x_j variables are forced to 1 by constraints (28) and (29) for both the MILP and its LP relaxation. By contrast, when $0 \leq z_{ij} < 1$, x_i or x_j must assume a value of 0 in the MILP, but not in the relaxation. Thus, in the LP relaxation, we can set more of the z variables to positive values than in the MILP. This raises the question of how many z variables we can set to 1 in the MILP. In the optimal solution, only z_{23} assumes a value of 1. So, can we set two of the z variables to 1 and find a feasible solution to the MILP? To answer this question, pick any two z variables and set them to 1. Since each z variable is involved in similar types of constraints, without loss of generality, we set z_{12} and z_{13} to 1. From the constraints:

$$\begin{aligned} z_{12} - x_1 &\leq 0 \\ z_{12} - x_2 &\leq 0 \\ z_{13} - x_1 &\leq 0 \\ z_{13} - x_3 &\leq 0 \end{aligned}$$

we see that x_1, x_2, and x_3 must all be set to 1. But this violates the constraint that the x variables can sum to at most 2. For any of the other two distinct pairs of z variables in this smaller model, all three x variables are forced to a value of 1 since for the MILP:

$$z_{ij} > 0 \Longleftrightarrow x_i = x_j = 1 \tag{32}$$

Thus, any distinct pair of z variables set to 1 forces three x variables to 1, violating the constraint that $x_1 + x_2 + x_3 \leq 2$. Hence, in any integer feasible solution, at most one z variable can be set to 1. This implies that the constraint:

$$z_{12} + z_{13} + z_{23} \leq 1$$

is a globally valid cut. And, we can see that it cuts off the optimal solution of the LP relaxation consisting of setting each z variable to 2/3.

We now generalize this to $(MILP)$, in which the x variables can sum to at most k. We wish to determine the number of z variables we can set to 1 in $(MILP)$ without forcing the sum of the x variables to exceed k. Suppose we set k of the x variables to 1. Since (32) holds for all pairs of x variables, without loss of generality, consider an integer feasible solution with $x_1 = x_2 = \cdots = x_k = 1$, and $x_{k+1} = \cdots = x_n = 0$. From (32), $z_{ij} = 1$ if and only if $1 \leq i \leq k$, $1 \leq j \leq k$, and $i < j$. We can therefore count the number of z variables that equal 1 when $x_1 = x_2 = \cdots = x_k = 1$. Specifically, there are $k(k-1)$ pairs (i, j) with $i \neq j$, but only half of them have $i < j$. So, at most $k(k-1)/2$ of the z_{ij} variables can be set to 1 when k of the x variables are set to 1. In other words,

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} z_{ij} \leq k(k-1)/2$$

is a globally valid cut.

Adding this cut to the instance with $n = 60$ and $k = 24$ enables CPLEX to solve the model to optimality in just over 2 hours and 30 minutes on the same machine using settings identical to those from the previous run without the cut. (See **Node Log #10**.) Note that the cut tightened the formulation significantly, as can be seen by the much better root node objective value of 4552.4000, which compares favorably to the root node objective value of 7640.4000 on the instance without the cut. Furthermore, the cut enabled CPLEX to add numerous zero-half cuts to the model that it could not with the original formulation. The zero-half cuts resulted in additional progress in the best node value that was essential to solving the model to optimality in a reasonable amount of time.

Node Log #10

```
        Nodes                                               Cuts/
   Node  Left     Objective  IInf  Best Integer     Best Node    ItCnt     Gap

*     0+    0                            0.0000                   1161     ---
      0     0     4552.4000   750        0.0000     4552.4000     1161     ---
*     0+    0                            6.0000     4552.4000     1161     ---
...
*     0+    0                         3477.0000     3924.7459    37882   12.88%
      0     2     3924.7459  1281     3477.0000     3924.7459    37882   12.88%
Elapsed real time =  51.42 sec. (tree size =  0.01 MB, solutions = 31)
      1     3     3919.3378  1212     3477.0000     3924.7459    39886   12.88%
      2     4     3910.8201  1243     3477.0000     3924.7459    42289   12.88%
      3     5     3910.8041  1144     3477.0000     3919.3355    44070   12.72%
...
 125571  7819        cutoff           3590.0000     3599.7046 60456851    0.27%
Elapsed real time = 9149.19 sec. (tree size = 234.98 MB, solutions = 43)
Nodefile size = 196.38 MB (168.88 MB after compression)
*126172  7231      integral     0     3591.0000     3599.7046 60571398    0.24%
 127700  5225        cutoff           3591.0000     3598.0159 60769494    0.20%
 131688     6        cutoff           3591.0000     3592.5939 60980430    0.04%

Zero-half cuts applied:  2244
```

```
Solution pool: 44 solutions saved.

MIP - Integer optimal solution:  Objective =  3.5910000000e+03
Solution time = 9213.79 sec.  Iterations = 60980442  Nodes = 131695
```

Given the modest size of the model, a run time of 2.5 hours to optimality suggests potential for additional improvements in the formulation. However, by adding one globally valid cut, we see a dramatic performance improvement nonetheless. Furthermore, the derivation of this cut draws heavily on the guidelines proposed for tightening the formulation. By using a small instance of the model, we can easily identify how removal of integrality restrictions enables the objective to improve. Furthermore, we use infeasibility to derive the cut: by recognizing that the simplified MILP model is infeasible when $z_{12} + z_{13} + z_{23} \geq 2$, we show that $z_{12} + z_{13} + z_{23} \leq 1$ is a valid cut.

5 Conclusion

Today's hardware and software allow practitioners to formulate and solve increasingly large and detailed models. However, optimizers have become less straightforward, often providing many methods for implementing their algorithms to enhance performance given various mathematical structures. Additionally, the literature regarding methods to increase the tractability of mixed integer linear programming problems contains a high degree of theoretical sophistication. Both of these facts might lead a practitioner to conclude that developing the skills necessary to successfully solve difficult mixed integer programs is too time consuming or difficult. This paper attempts to refute that perception, illustrating that practitioners can implement many techniques for improving performance without expert knowledge in the underlying theory of integer programming, thereby enabling them to solve larger and more detailed models with existing technology.

Acknowledgements

Dr. Klotz wishes to acknowledge all of the CPLEX practitioners over the years, many of whom have provided the wide variety of models that revealed the guidelines described in this paper. He also wishes to thank the past and present CPLEX development, support, and sales and marketing teams who have contributed to the evolution of the product. Professor Newman wishes to thank former doctoral students Chris Cullenbine, Brian Lambert, Kris Pruitt, and Jennifer Van Dinter at the Colorado School of Mines for their helpful comments; she also wishes to thank her colleagues Jennifer Rausch (Jeppeson, Englewood, Colorado) and Professor Josef Kallrath (BASF-AG, Ludwigshafen, Germany) for helpful comments on an earlier draft. Both authors thank an anonymous referee for his helpful comments that improved the paper.

Both authors wish to remember Lloyd Clarke (February 14, 1964-September 20, 2007). His departure from the CPLEX team had consequences that extended beyond the loss of an important employee and colleague.

References

Aardal, K. and Lenstra, A., 2004. "Hard equality constrained integer knapsacks." *Mathematics of Operations Research*, **3**(29): 724–738.

Achterberg, T., 2007. *Constraint Integer Programming (Ph.D. Dissertation)*, Technical University Berlin, Berlin.

Balas, E., 1965. "An additive algorithm for solving linear programs with zero-one variables." *Operations Research*, **13**(4): 517–546.

Balas, E., 1998. "Disjunctive programming: Properties of the convex hull of feasible points." *Discrete Applied Mathematics Tech. Report MSRR 348, Carnegie Mellon University*, **89**(1-3): 3–44.

Bertsimas, D. and Stock Patterson, S., 1998. "The air traffic flow management problem with enroute capacities." *Operations Research*, **46**(3): 406–422.

Bixby, R. and Rothberg, E., 2007. "Progress in computational mixed integer programming – a look back from the other side of the tipping point." *Annals of Operations Research*, **149**(1): 37–41.

Camm, J., Raturi, A. and Tadisina, S., 1990. "Cutting big M down to size." *Interfaces*, **20**(5): 61–66.

Cornuejols, G., Urbaniak, R., Weismantel, R. and Wolsey, L., 1997. "Decomposition of integer programs and of generating sets." *Lecture Notes in Computer Science; Proceedings of the 5th Annual European Symposium on Algorithms*, **1284**: 92–102.

Danna, E., 2008. "Performance variability in mixed integer programming." Presentation at MIP 2008 Workshop, Columbia University.

FICO, 2012. *Xpress-MP Optimization Suite*, Minneapolis, MN.

Grötschel, ed., 2004. *The Sharpest Cut: The Impact of Manfred Padberg and His Work*, MPS-SIAM Series on Optimization.

Gurobi, 2012. *Gurobi Optimizer*, Houston, TX.

IBM, 2012. *ILOG CPLEX*, Incline Village, NV.

Klotz, E. and Newman, A., To appear. "Practical guidelines for solving difficult linear programs." *Surveys in Operations Research and Management Science*, **doi:10.1016/j.sorms.2012.11.001**.

Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R., Danna, E., Gamrath, G., Gleixner, A., Heinz, S., Lodi, A., Mittelmann, H., Ralphs, T., Salvagnin, D., Steffy, D. and Wolter, K., 2011. "MIPLIB 2010." *Mathematical Programming Computation*, **3**(2): 103–163.

Lambert, W., Brickey, A., Newman, A. and Eurek, K., to appear. "Open pit block sequencing formulations: A tutorial." *Interfaces*.

MOPS, 2012. *MOPS*, Paderborn, Germany.

MOSEK, 2012. *MOSEK Optimization Software*, Copenhagen, Denmark.

Rardin, R., 1998. *Optimization in Operations Research*, Prentice Hall, chap. 6.

Rebennack, S., Reinelt, G. and Pardalos, P., 2012. "A tutorial on branch and cut algorithms for the maximum stable set problem." *International Transactions in Operational Research*, **19**(1-2): 161–199.

Savelsbergh, M., 1994. "Preprocessing and probing techniques for mixed integer programming problems." *INFORMS Journal on Computing*, **6**(4): 445–454.

Watters, L., 1967. "Reduction of integer polynomial programming to zero-one linear programming problems." *Operations Research*, **15**(6): 1171–1174.

Winston, W., 2004. *Operations Research: Applications and Algorithms*, Brooks/Cole, Thompson Learning.

Wolsey, L., 1998. *Integer Programming*, Wiley.