An exact rational mixed-integer programming solver

William Cook*[1], Thorsten Koch[2], Daniel E. Steffy*[1], and Kati Wolter †[2]

[1] School of Industrial and Systems Engineering,
Georgia Institute of Technology, Atlanta, GA,
{bico,dsteffy}@isye.gatech.edu

[2] Zuse Institute Berlin, Germany, {koch,wolter}@zib.de

November 15, 2010

Abstract

We present an exact rational solver for mixed-integer linear programming which avoids the numerical inaccuracies inherent in the floating-point computations adopted in existing software. This allows the solver to be used for establishing fundamental theoretical results and in applications where correct solutions are critical due to legal and financial consequences. Our solver is a hybrid symbolic/numeric implementation of LP-based branch-and-bound, using numerically-safe bounding methods for all binding computations in the search tree. Computing provably accurate solutions by dynamically choosing the fastest of several available methods depending on the structure of the instance, our exact solver is only moderately slower compared to an inexact floating-point branch-and-bound solver. The software is incorporated into the SCIP optimization framework, using the exact LP solver QSopt_ex and the GMP arithmetic library. Computational results are presented for a suite of test instances taken from the Miplib and Mittelmann collections.

*Research supported by NSF Grant CMMI-0726370 and ONR Grant N00014-08-1-1104.
†Research funded by the DFG Priority Program 1307 "Algorithm Engineering".

1 Introduction

Mixed-integer programming (MIP) is a powerful and flexible tool for modeling and solving decision problems. Software based on these ideas is used in many application areas. Despite their widespread use, few available software packages provide any guarantee of correct answers or certification of results. Possible inaccuracy is caused by the use of floating-point numbers [14]. Floating-point calculations necessitate the use of built-in tolerances for testing feasibility and optimality, and can lead to calculation errors in the solution of linear-programming (LP) relaxations and in the methods used for creating cutting planes to improve these relaxations.

Due to a number of reasons, for many industrial MIP applications near optimal solutions are sufficient. CPLEX, for example, defaults to a relative MIP optimality tolerance of 0.001. Moreover, when data describing a problem arises from imprecise sources, exact feasibility is usually not necessary. Nonetheless, accuracy is important in many settings. Direct examples arise in the use of MIP models to establish fundamental theoretical results and in subroutines for the construction of provably accurate cutting planes. Furthermore, industrial customers of MIP software request modules for exact solutions in critical applications. Such settings include the following.

- Feasibility problems, including chip verification in the VLSI design process [1].
- Compiler optimization, including instruction scheduling [23] and register allocation [15].
- Combinatorial auctions [10], where serious legal and financial consequences can result from incorrect solutions.

Exact methods are also important as a standard of comparison for developing efficient, but possibly inaccurate, methods.

Optimization software relying exclusively on exact rational arithmetic has been observed to be prohibitively slow, motivating the development of more sophisticated techniques to compute exact solutions. Significant progress has been made recently toward computationally solving LP models exactly over the rational numbers using hybrid symbolic/numeric methods [6, 11, 12, 17, 18], including the release of the software package QSopt_ex [7]. Exact MIP has seen less computational progress than exact LP, but significant first steps have been taken. An article by Neumaier and Shcherbina [20] describes methods for safe MIP computation, including strategies for generating safe LP bounds, infeasibility certificates, and cutting planes. The methods they describe involve directed rounding and interval arithmetic with floating point numbers to avoid incorrect results.

The focus of this paper is to introduce a hybrid branch-and-bound approach for exactly solving MIPs over the rational numbers. Section 2 describes how rational and safe floating-point computation can be coupled together, providing a fast and general framework for exact computation. Section 3 describes several methods for computing valid LP bounds, which is a critical component of the hybrid approach. Section 4 describes an exact branch-and-bound implementation within SCIP and includes detailed computational results on a range of test libraries comparing different dual bounding strategies. The exact solver is compared with an inexact floating-point branch-and-bound solver and observed to be only moderately slower.

2 Hybrid Rational/Safe Floating-Point Approach

Two ideas for exact MIP that have been proposed in the literature, and tested to some extent, are the *pure rational approach* [6] and the *safe floating-point (FP) approach* [9, 20]. Both utilize LP-based branch-and-bound. The difference lies in how they ensure the computed results are correct.

In the pure rational approach, correctness is achieved by storing the input data as rational numbers, by performing all arithmetic operations over the rationals, and by applying an exact LP solver [12] in the dual bounding step. What makes this approach interesting is that it can handle a broad problem class: MIP instances that are given by rational data. Furthermore, since LP relaxations are solved exactly, branch-and-bound nodes that can be cut off due to bounding or infeasibility will be recognized immediately, preventing unnecessary branching. However, replacing all FP operations by rational computation will increase running times noticeably. For example, although the exact LP solver QSopt_ex avoids many unnecessary rational computations and is efficient on average, Applegate et al. [6] observed an exact MIP solver which called QSopt_ex for each LP computation to be two or three orders of magnitude slower than commercial MIP solvers.

In order to limit the degradation in running time, the idea of the safe-FP approach is to continue to use FP-numbers as much as possible, particularly within the LP solver. However, to ensure correct decisions in the branch-and-bound algorithm extra work is necessary. This consists of controlling the rounding mode of crucial arithmetic operations, like the primal feasibility test for an approximate LP solution, and in post-processing the approximate LP solution to obtain a correct dual bound. Such a safe dual bounding technique has been successfully implemented in Concorde [5] for the traveling salesman problem. A generalization of the method for MIPs can be found in [20]. Furthermore, the idea of manipulating the rounding mode can be applied to *cutting-plane separation*. In [9], this idea was used to generate numerically safe Gomory mixed-integer cuts. Nevertheless, whether the safe-FP approach leads to acceptable running times for general MIPs has not been investigated.

Although the safe-FP version of branch-and-bound has great advantages in speed over the pure rational approach, it has several clear disadvantages. Everything, including input data and primal solutions, is stored as FP numbers. Therefore, correct results can only be ensured for MIP instances that are given by FP-representable data and that have a FP-representable optimal solution if they are feasible. Some rationally defined problems can be scaled to have FP-representable data. However, this is not always possible due to the limited representation of floating-point numbers, and the resulting large coefficients can lead to numerical difficulties. The applicability is even further limited as the safe dual bounding method discussed in [20] requires, in general, lower and upper bounds on all variables. Moreover, when dealing with MIPs some branch-and-bound nodes may only be processable by an exact LP solver. One such situation occurs when the current subproblem is just an LP (all integer variables have been fixed), i.e., branching is not possible, but post-processing its approximate LP solution does not allow the node to be cut off. To immediately process this node, the LP must be solved exactly.

The pure rational approach is always applicable but introduces a large overhead in running time while the safe-FP approach is more efficient but of limited applicability. Since we want to exactly and efficiently solve MIPs that are given by rational data we have developed a version of branch-and-bound that attempts to combine the advantages of the pure rational and safe-FP approaches, and to compensate for their individual weaknesses. The idea is to work with two branch-and-bound processes. The *main process* implements the rational approach. Its result is

surely correct and will be issued to the user. The other one serves as a *slave process*. Here, the faster safe-FP approach is applied. To achieve reasonable running time whenever possible the expensive rational computation of the main process will be skipped and certain decisions from the faster safe-FP process will be substituted. In particular, dual bound computations will be avoided as they require exact LP solves in the rational process. Instead, dual bounds will be taken from the safe-FP process. However, due to the rational process, the exact problem data are known, primal solutions can be stored correctly, and exact LP solves are always possible.

The complete procedure is given in Algorithm 1. The set of floating-point representable numbers is denoted by $\mathbb{M}$, $\underline{x} \in \mathbb{M}$ and $\overline{x} \in \mathbb{M}$ stand for lower and upper approximations of $x \in \mathbb{Q}$, respectively. The slave process, which utilizes the safe-FP approach, works on a MIP instance with FP-representable data. It is set up in Step 1 of the algorithm. If the input data are already FP-representable, both processes solve the same MIP instance, i.e., $\widetilde{P} := P$ and $\widetilde{c} := c$ in Step 1. In principle, this results in employing only the safe-FP approach except for some necessary exact LP solves. Otherwise, an approximation of the MIP with $P \approx \widetilde{P}$, $c \approx \widetilde{c}$ or a relaxation with $P \subseteq \widetilde{P}$, $c = \widetilde{c}$ is constructed; called *FP-approximation* and *FP-relaxation*, respectively. The choice depends on which dual bounding method we want to apply in the slave process, one of which requires an FP-relaxation. Generating dual bounds via relaxations can lead to weaker bounds, in particular, if equalities are present. Still, a larger branch-and-bound tree might be processed faster through this method. We also remark that any optimal solution identified by this procedure is guaranteed to have a rational description, despite the fact that some feasible/optimal solutions can have irrational components.

On the implementation side, we maintain only a single branch-and-bound tree. At the root node of this common tree, we store the LP relaxations of both processes: $\max\{c^T x : x \in LP\}$ and $\max\{\widetilde{c}^T x : x \in \widetilde{LP}\}$. In addition, for each node, we know which branching constraint was added to create both subproblems. Branching on variables, performed in Step 8, introduces the same bounds for both processes. They are FP-representable which means an FP-relaxation property will be preserved, i.e., throughout the tree, $P_j \subseteq \widetilde{P}_j$ for corresponding subproblems if $P \subseteq \widetilde{P}$ holds.

The use of primal and dual bounds to discard subproblems (see Steps 5, 6, and 7) is a central component of the branch-and-bound algorithm. Therefore, in the exact MIP setting, the efficiency strongly depends on the strength of the dual bounds and the time spent generating them (Step 5). The starting point of this step is the *approximate LP solution* of the slave process. It is obtained by an LP solver that works on FP-numbers and accepts rounding errors; referred to as *inexact LP solvers*. Depending on the result, we safely check whether the rational LP, i.e., the exact LP relaxation, is also infeasible or we compute a safe dual bound by post-processing the approximate LP solution. When working with an FP-approximation in the slave process we need to access the rational data from the main process here. Different techniques are discussed in Section 3 and are computationally evaluated in Section 4. We only perform the exact but expensive dual bound computation of the main process if it is necessary (Step 5(a)ii).

Dual and primal bounds are stored as FP-numbers and the bounding in Step 6 is performed without tolerances; a computed bound that is not FP-representable is relaxed in order to be safe. For the primal (lower) bound L, this means $L < c^{\text{MIP}}$ if the objective value c^{MIP} of the incumbent solution x^{MIP} is not in $\mathbb{M}$.

Algorithm 1 identifies primal solutions by checking LP solutions for integrality. This check, performed in Step 7, depends on whether the LP was already solved exactly at the current node. If so, we exactly test the integrality of the exact LP solution. Otherwise, we decide if it

Algorithm 1 Branch-and-bound for exactly solving MIPs

Input: (MIP) $\max\{c^T x : x \in P\}$ with $P := \{x \in \mathbb{R}^n : Ax \leq b, x_i \in \mathbb{Z} \text{ for all } i \in I\}$, $A \in \mathbb{Q}^{m\times n}$, $b \in \mathbb{Q}^m$, $c \in \mathbb{Q}^n$, and $I \subseteq \{1, \ldots, n\}$.

Output: *Exact* optimal solution $x^\star$ of MIP with objective value $c^\star$ or conclusion that MIP is *truly* infeasible ($c^\star = -\infty$).

1. *FP-problem:* Store (FP-MIP) $\max\{\widetilde{c}^T x : x \in \widetilde{P}\}$ with $\widetilde{P} := \{x \in \mathbb{R}^n : \widetilde{A}x \leq \widetilde{b}, x_i \in \mathbb{Z} \text{ for all } i \in I\}$, $\widetilde{A} \in \mathbb{M}^{m\times n}$, $\widetilde{b} \in \mathbb{M}^m$, and $\widetilde{c} \in \mathbb{M}^n$.
2. *Init:* Set $\mathcal{L} := \{(P, \widetilde{P})\}$, $L := -\infty$, x^{MIP} to be empty, and $c^{\text{MIP}} := -\infty$.
3. *Abort:* If $\mathcal{L} = \emptyset$, stop and return x^{MIP} and c^{MIP}.
4. *Node selection:* Choose $(P_j, \widetilde{P}_j) \in \mathcal{L}$ and set $\mathcal{L} := \mathcal{L} \setminus \{(P_j, \widetilde{P}_j)\}$.
5. *Dual bound:* Solve LP-relaxation $\max\{\widetilde{c}^T x : x \in \widetilde{LP}_j\}$ *approximately.*
   (a) If $\widetilde{LP}_j$ is *claimed* to be empty, *safely* check if LP_j is empty.
      i. If LP_j is empty, set $c^\star := -\infty$.
      ii. If LP_j is not empty, solve LP-relaxation $\max\{c^T x : x \in LP_j\}$ *exactly.* Let $x^\star$ be an *exact* optimal LP solution and $c^\star$ its objective value.
   (b) If $\widetilde{LP}_j$ is *claimed* not to be empty, let $x^\star$ be an *approximate* optimal LP solution and compute a *safe* dual bound $c^\star$ with $\max\{c^T x : x \in LP_j\} \leq c^\star$.
6. *Bounding:* If $\overline{c^\star} \leq L$, goto Step 3.
7. *Primal bound:*
   (a) If $x^\star$ is *approximate* LP solution and claimed to be feasible for FP-MIP, solve LP-relaxation $\max\{c^T x : x \in LP_j\}$ *exactly.* If LP_j is *in fact* empty, goto Step 3. Otherwise, let $x^\star$ be an *exact* optimal LP solution and $c^\star$ its objective value.
   (b) If $x^\star$ is *exact* LP solution and *truly* feasible for MIP, set $x^{\text{MIP}} := x^\star$, $c^{\text{MIP}} := c^\star$, and $L := \underline{c^\star}$ and goto Step 3.
8. *Branching:* Choose index $i \in I$ with $x_i^\star \notin \mathbb{Z}$.
   (a) Split P_j in $Q_1 := P_j \cap \{x : x_i \leq \lfloor x_i^\star \rfloor\}$, $Q_2 := P_j \cap \{x : x_i \geq \lceil x_i^\star \rceil\}$.
   (b) Split $\widetilde{P}_j$ in $\widetilde{Q}_1 := \widetilde{P}_j \cap \{x : x_i \leq \lfloor x_i^\star \rfloor\}$, $\widetilde{Q}_2 := \widetilde{P}_j \cap \{x : x_i \geq \lceil x_i^\star \rceil\}$.

   Set $\mathcal{L} := \mathcal{L} \cup \{(Q_1, \widetilde{Q}_1), (Q_2, \widetilde{Q}_2)\}$ and goto Step 3.

is worth solving the LP exactly by checking if the approximate LP solution is nearly integral. In this case, we solve the LP exactly, using the corresponding basis to warm-start the exact LP solver (hopefully with few pivots and no need to increase the precision) and perform the exact integrality test on the exact LP solution. In order to correctly report the optimal solution found at the end of Step 3, the incumbent solution (that is, the best feasible MIP solution found thus far) and its objective value are stored as rational numbers.

The hybrid approach can be extended to a branch-and-cut algorithm with primal heuristics and presolving; but the focus of this paper is on the development of the basic framework.

3 Safe Dual Bound Generation Techniques

This section describes several methods for computing valid LP dual bounds. The overall speed of the MIP solver will be influenced by several aspects of the dual bounding strategy; how generally applicable the method is, how fast the bounds can be computed and how strong the bounds are, because weak bounds lead to an increased node count.

3.1 Exact LP Solutions

The most straightforward way to compute valid LP bounds is to solve each node LP relaxation exactly. This strategy is always applicable and produces the tightest possible bounds. Within a branch-and-bound framework the dual simplex algorithm can be warm started with the final basis computed at the parent node, speeding up the LP solution process. The fastest exact rational LP solver currently available is QSopt_ex [6]. The strategy used by this solver can be summarized as follows: the basis returned by a double-precision LP solver is tested for optimality by symbolically computing the corresponding basic solution, if it is suboptimal then additional simplex pivots are performed with an increased level of precision and this process is repeated until the optimal basis is identified. This method is considerably faster than using rational arithmetic exclusively and is usually no more than two to five times slower than inexact floating-point LP solvers. For problems where the basis determined by the double-precision subroutines of QSopt_ex is not optimal, the additional increased precision simplex pivots and additional exact basic solution computations significantly increase the solution time.

3.2 Verify LP Basis

Any exactly feasible dual solution provides a valid dual bound. Therefore, valid dual bounds can be determined by symbolically computing the dual solution corresponding to a numerically obtained LP basis, without performing the extra steps required to identify the exact optimal solution. If the dual solution is feasible, its objective value gives a valid bound. If it is infeasible, an infinite bound is returned. Within the branch-and-bound algorithm, an infinite dual bound will lead to more branching. Due to the fixing of variables, branching often remediates numerical problems in the LP relaxations. This strategy avoids the extended precision simplex pivoting that can occur when solving each node LP exactly, but it can increase the number of nodes.

3.3 Primal-Bound-Shift

Valid bounds can also be produced by correcting approximate dual solutions. A special case occurs when all primal variables have finite upper and lower bounds. This technique was em-

ployed by Applegate et. al. in the CONCORDE software package [5] and is described more generally for MIPs by Neumaier and Shcherbina [20]. Consider a primal problem of the form $\max\{c^T x : Ax \leq b, 0 \leq x \leq u\}$ with dual $\min\{b^T y + u^T z : A^T y + z \geq c,\ y, z \geq 0\}$. Given an approximate dual solution $\tilde{y}, \tilde{z}$, an exactly feasible dual solution $\hat{y}, \hat{z}$ is constructed by setting $\hat{y}_i = \max\{0, \tilde{y}_i\}$ and $\hat{z}_i = \max\{0, (c - A^T \hat{y})_i\}$. This gives the valid dual bound $b^T \hat{y} + u^T \hat{z}$. This bound can be computed using floating-point arithmetic with safe directed rounding to avoid the symbolic computation of the dual feasible solution. The simplicity of computing this bound suggests it will be an excellent choice when applicable. However, if some primal variable bounds are large or missing this procedure can produce weak or infinite bounds, depending on the feasibility of $\tilde{y}, \tilde{z}$.

3.4 Project and Shift

Correcting an approximate dual solution to be exactly feasible in the absence of primal variable bounds is still possible. Consider a primal problem of the form $\max\{c^T x : Ax \leq b\}$ with dual $\min\{b^T y : A^T y = c,\ y \geq 0\}$. An approximate dual solution $\tilde{y}$ can be corrected to be feasible by projecting it into the affine hull of the dual feasible region and then shifting it to satisfy all of the non-negativity constraints, while maintaining feasibility of the equality constraints. These operations could involve significant computation if performed on a single LP. However, under some assumptions, the most time consuming computations can be performed only once at the root node of the branch-and-bound tree and reused for each node bound computation. The root node computations involve solving an auxiliary LP exactly and symbolically LU factoring a matrix; the cost of each node bound computation is dominated by performing a back-solve of a pre-computed symbolic LU factorization, which is often faster than solving a node LP exactly. A detailed description and computational study of this algorithm can be found in [22]. A related method is also described by Althaus and Dumitriu [4].

3.5 Combinations and Beyond

The ideal dual bounding method is generally applicable, produces tight bounds, and computes them quickly. Each of the four methods described so far represent some trade-off between these conflicting characteristics. The exact LP method is always applicable and produces the tightest possible bounds, but is computationally expensive. The primal-bound shift method computes valid bounds very quickly, but relies on problem structure that may not always be present. The basis verification and project and shift methods provide compromises in between, with respect to speed and generality. Therefore, a strategy that combines and switches between these bounding techniques is the best choice for an exact MIP solver intended to efficiently solve a broad class of problems. Also, the relative performance of these dual bounding methods strongly depends on the problem structure, and this can change throughout the tree.

In Section 4, we will evaluate the performance of each dual bounding method and analyze in what situations which technique works best. In a final step, we will study different strategies to automatically decide how to compute safe dual bounds for a given MIP instance. The central idea is to apply fast primal-bound shift as often as possible and if necessary employ another method depending on the problem structure. In this connection, we will address the question of whether this decision should be static or dynamic. The dual bounding method can be selected at the beginning, in Step 1, and remain fixed throughout the tree. This allows us to work with FP-approximations whenever we do not select primal-bound shift. Alternatively, we can decide

on the method dynamically at every node, in Step 5. The drawback is that this requires an FP-relaxation for the slave process in order to support fast primal-bound shift at every node. However, how do we decide on primal-bound shift: by guessing or testing? This is addressed in Section 4. Beyond that, we will analyze whether it is a good idea to compute safe dual bounds only if they are really required, i.e., at nodes where the unsafe bound would lead to pruning. Furthermore, we experiment with interleaving our actual selection strategy with exact LP solves to eliminate special cases where weak bounds cause the solver to keep branching in subtrees that would have been cut off by the exact LP bound.

4 Computational Study

In this section, we investigate the performance of our exact MIP framework employing the different safe dual bounding techniques discussed above: primal-bound shift ("BoundShift"), project and shift ("ProjectShift"), basis verification ("VerifyBasis"), and exact LP solutions ("ExactLP"). We will first look at each method on its own and then examine them within the branch-and-bound algorithm. At the end, we discuss and test a strategy to automatically switch between the most promising bounding techniques ("Auto") as well as variants where we only guess whether primal-bound shift will work ("Auto-Static"), where we interleave the normal selection with exact LP calls ("Auto-Ileaved"), and where we restrict safe dual bound generation to the necessary nodes ("Auto-Limited").

The discussed algorithms were implemented into the branch-and-bound algorithm provided by the MIP framework SCIP 1.2.0.8 [1, 2, 21], using best bound search for node selection and first fractional variable branching. To solve LPs approximately and exactly we call CPLEX 12.2 [16] and QSOPT_EX 2.5.5 [7], respectively. Rational computations are based on the GMP library 4.3.1 [13]. All benchmark runs were run on Intel E5420 CPUs with 4 cores and 16 GB RAM each. To maintain accurate results only one computation was run at the same time. We imposed a time limit of 1 hour and a memory limit of 3 GB. Our test set contains all instances of the MIPLIB 3.0 [8] and MIPLIB 2003 [3] libraries and from the Mittelmann collections [19]. We excluded eight instances that could not be run within the imposed limits. This gives a test suite of 155 MIP instances.

We start with evaluating the root node behavior of the dual bounding methods. Our performance measures are: time overhead and bound quality. Table 1 presents for each method the additional time spent to compute dual bounds safely relative to "BoundShift" in shifted geometric mean[1] ("DBT"). Note that the timings used to measure the computation times are always rounded up to one second if they are smaller. Furthermore, Table 1 states the number of instances for which a certain dual bound quality was achieved. The quality is given by the relative difference between the computed safe dual bound $c^\star$ and the exact LP value $c^{\star\star} := \max\{c^T x : x \in LP_j\}$. However, we actually compare the FP-representable upper approximations of both values, as used in Algorithm 1, and define the relative difference as $d := (\overline{c^\star} - \overline{c^{\star\star}})/\max\{1, |\overline{c^\star}|, |\overline{c^{\star\star}}|\}$. The corresponding columns in Table 1 are: "Zero" difference for $d = 0$, "S(mall)" difference for $d \in (0, 10^{-9}]$, "M(edium)" difference for $d \in (10^{-9}, 10^{-3}]$, and "L(arge)" difference for $d \in (10^{-3}, \infty)$. Column "∞" counts the worst case behavior, i.e., infinite dual bounds. Note that for these tests we doubled the time limit in order to leave enough room to compute the exact dual bound value $c^{\star\star}$.

[1]The shifted geometric mean of $t_1, \ldots, t_n$ is defined by $(\Pi_{i=1}^n (t_i + s))^{1/n} - s$ and is an intermediate measure between the arithmetic and geometric means, we use a shift factor of $s = 10$ for time and $s = 100$ for nodes.

Setting	Zero	S	M	L	∞	DBT
BoundShift	37	72	3	0	43	1.0
ProjectShift	51	85	10	4	5	7.1
VerifyBasis	151	2	0	0	2	1.9
ExactLP	155	0	0	0	0	3.0
Auto	53	96	5	1	0	1.2
Auto-Static	54	95	5	1	0	1.4
Auto-Ileaved	53	96	5	1	0	1.2

Table 1. Root node dual bound: Relative difference to exact bound and additional computation time in shifted geometric mean.

		optimal (24)			timeout (105)		timeout+gap (21)		
Setting	slv	Nodes	Time	DBT	Nodes	DBT	Nodes	Gap	DBT
BoundShift	33	6753	39.0	5.6	475948	545.7	1509219	70.3	512.9
ProjectShift	35	6097	103.4	63.9	87640	2569.6	199486	18.9	2911.8
VerifyBasis	33	6073	126.3	88.4	57442	2748.4	167485	19.9	3038.9
ExactLP	30	6072	146.5	109.6	40452	3073.2	134083	20.4	3163.1
Inexact	50	6051	29.1	—	969872	—	2477629	9.5	—
Auto	43	6087	38.9	8.9	281844	967.1	501730	16.2	1241.7
Auto-Static	42	6087	43.4	11.1	268833	978.2	505212	16.2	1240.6
Auto-Ileaved	45	6066	39.5	9.9	273331	1014.2	504037	15.7	1399.0
Auto-Limited	37	7395	35.2	5.5	507565	369.8	1109675	483.7	660.6

Table 2. Overall performance. "slv" is number of instances solved, "DBT" is additional time spent computing exact dual bounds.

As expected, primal-bound shift is the fastest method, but it fails to produce dual bounds on 43 instances in contrast to only 5 for project and shift and only 2 for basis verification. This is, project and shift meets its requirements most of the time and the bases obtained by CPLEX are almost always dual feasible and even optimal. Still, primal-bound shift provides very good quality bounds if it works; most of these instances fall into the "Zero" and "S(mall)" categories. When primal-bound shift works we expect to obtain strong bounds and whenever it fails we assume basis verification or project and shift to be applicable. Which method to chose depends on the time overhead. However, this can not be decided by looking at the root node as project and shift involves an expensive initial set-up step. This can be seen in the performance profile in Figure 1. It visualizes the relative overhead times for the safe dual bounding methods. For each of them, it plots the number of instances for which the dual bounding step was performed within a given factor of the bounding time of the fastest method.

Where basis verification is in most cases only up to 10 times slower than the fastest method primal-bound shift, project and shift can be up to 100 times slower at the root node because of the set-up step. But we will see that this overhead often pays off in the branch-and-bound tree. Let us further remark that the idea of basis verification, to omit the repair step of exact LP solvers, actually limits the occasional large overhead of QSOPT_EX. As it produces very good bounds we will prefer basis verification to exact LP solves in our automatic dual bound selection strategy. Comparing their branch-and-bound behavior leads to the same conclusion.

We will now analyze the effect of the dual bound methods on the *overall performance* of the exact MIP solver and compare it with the inexact branch-and-bound version of SCIP. Table 2, reports the number of instances that were solved within the imposed limits (Column "slv"), for each setting. 50 instances were solved in at least one test run, indeed all of them by at least the inexact version "Inexact". Figure 2 gives a detailed comparison of the solution times for these 50 instances. For a setting where an instance had a timeout it is reported with an infinite ratio to the fastest setting. Thus, the points at the right boarder of the graphic reflect the "slv" column. To compare the methods on all instances of the test set, we split them into three groups with different performance measures each. The first group "optimal" contains all 24 instances that were *solved by all versions*. For this group, we present in Table 2, for each setting, the number of branch-and-bound nodes "Nodes", the solution times "Time" in seconds, and the additional time spent in the safe dual bounding step "DBT", all in shifted geometric mean. The second group "timeout" includes all instances where *every test run had a timeout*. Here, the number of branch-and-bound nodes "Nodes" indicates how fast a method is. In general, the more nodes

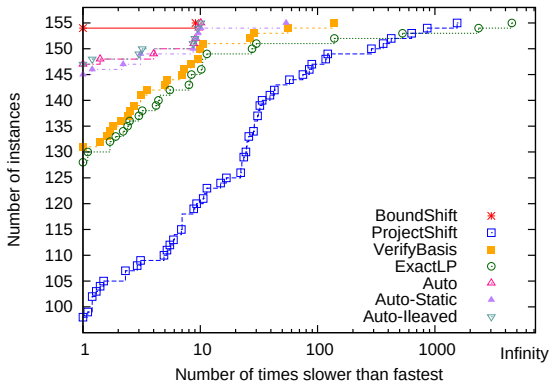

Figure 1. Additional time for root node dual bound computations.

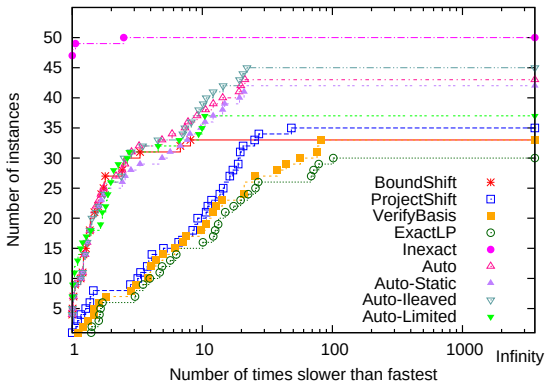

Figure 2. Overall solving times.

processed, the faster the dual bounding method. For a subgroup of 21 instances, we had a *finite primal-dual gap in every test run*. This gives our third group "timeout+gap" for which we also report the primal-dual gaps in shifted geometric mean "Gap".

The observations made for the root node carry forward to the application in the branch-and-bound algorithm. Primal-bound shift leads to the fastest node processing. Basis verification has a slightly better performance than solving LPs exactly at every node. However, basis verification is often outperformed by project and shift.

A closer look reveals that primal-bound shift introduces not only a small overhead at the root but throughout the tree. On the "optimal" group, it performs similar to the inexact code in mean. For the individual instances, we usually experience a slow-down of at most 2. The few large slow-down factors of up to 10 can, for example, be explained by a slight node increase due to a small number of missing bounds and by expensive exact LP calls to prove infeasibility of the node or primal feasibility of the basic LP solution. A similar decrease in running time can be observed for the "timeout" group via the node count. However, this group includes instances where primal-bound shift fails: the gap in geometric mean is 70.3% on the "timeout+gap" group in contrast to 9.5% for the inexact code.

In contrast, project and shift gives a much better gap (18.9% in mean) on these instances. That is, the often observed good dual bound quality at the root node seems to stay throughout the tree. We already pointed out the big overhead this costs at the root node. However, the method is fast on the rest of the tree and leads to an acceptable slow-down compared to the inexact code. In mean, it is only 3 times slower on the "optimal" group and only 10 times on the "timeout" instances. In this fashion, most of the 35 instances solved by this setting are only up to 10 times slower than the inexact code. If we compare project and shift with basis verification we see a similar and often better performance for project and shift. For example, it solves two more instances. However, on some instances basis verification works better. We tested different problem characteristics and found the number of non-zeros in the constraint matrix to be a good criterium for choosing between project and shift and basis verification. In the automatic dual bound selection, we prefer project and shift as long as the matrix has at most $10,000$ non-zeros.

We already gave some remarks concerning a strategy that automatically chooses a dual bounding method. Another important observation for this purpose is that replacing FP-approximations by FP-relaxations does not affect the performance much: running project and shift on a FP-relaxation gave similar results. Therefore, we decided to always set up

an FP-relaxation in Step 1. This way, we are allowed to apply primal-bound shift whenever we want to. The automatic decision process used in the "Auto" run works as follows. At every node, we first test whether primal-bound shift produces a finite bound. If not, we choose primal-bound shift or basis verification depending on the structure of the constraint matrix as explained above.

The results show that we actually combined the advantages of all dual bounding methods. We can solve all 35 instances that primal-bound shift solved as well as 8 additional instances by automatically switching to other dual bounding methods at the nodes. Furthermore, on the "timeout+gap" group, where project and shift produced a much better gap than primal-bound shift, the automatic setting processes in mean even two times more nodes than project and shift and this way further improves the gap.

In Section 3.5, we discussed three possible improvements for the automatic dual bound selection procedure. The first one, to only guess whether primal-bound shift will work, is implemented in test run "Auto-Static". The guess is static, i.e., does not change throughout the tree. We skip primal-bound shift if more than 20% of the variables have lower or upper bounds with absolute value larger than 10^6. Comparing both automatic settings shows that it is no problem to actually test whether primal-bound shift works and it even leads to a slightly better performance. The second idea was to interleave the strategy with exact LP calls whenever the node is very likely to be cut off. The test run ("Auto-Ileaved") showed that this applies not very often but when it does apply it actually helps. We solved two more instances to optimality and improved the gap on the "timeout+gap" group.

The third extension suggested in Section 3.5 was to only compute bounds safely if they are actually used for a crucial decision, i.e., if the unsafe bound with tolerances would lead to cutting off a node. Looking at the overall behavior for the corresponding test run "Auto-Limited", it is not clear whether this is a good idea in general. For one thing, the node count on the "timeout" instances doubles, i.e., the node processing is much faster on average. On the other hand, we can solve only 37 instead of 43 instances. We cannot tell anything about the quality on the rest of the instances as the primal-dual-gap does not improve steadily. Further testing is needed here, e.g., by applying safe dual bounding steps at selected levels of the tree.

5 Conclusion

From these computational results we can make several key observations. Each dual bounding method studied has strengths and weaknesses depending on the problem structure. Automatically switching between these methods in a smart way solves more problems than any single dual bounding method on its own. Of the 50 problems solved within one hour by the floating-point branch-and-bound solver, 45 could also be solved exactly and the solution time was usually no more than 10 time slower. This demonstrates that the hybrid methodology can lead to an efficient exact branch-and-bound solver, not limited to specific classes of problems. As our focus has been exclusively on the branch-and-bound procedure, we have compared the exact solver against a floating-point solver restricted to pure branch-and-bound. The exact solver is still not directly competitive with the full version of SCIP with its additional features enabled. However, it is realistic to think that the future inclusion of additional MIP machinery including cutting planes, more extensive presolving and heuristics into this exact framework could lead to a fully functioning exact MIP solver that is not prohibitively slower than its inexact counterparts.

References

[1] Tobias Achterberg. *Constraint Integer Programming*. Ph.D. thesis, Technische Universität Berlin, 2007.

[2] Tobias Achterberg. SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1:1–41, 2009.

[3] Tobias Achterberg, Thorsten Koch, and Alexander Martin. The mixed integer programming library: MIPLIB 2003. `http://miplib.zib.de`.

[4] Ernst Althaus and Daniel Dumitriu. Fast and accurate bounds on linear programs. In Jan Vahrenhold, editor, *Experimental Algorithms*, volume 5526 of *Lecture Notes in Computer Science*, pages 40–050. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-02011-7_6.

[5] David L. Applegate, Robert E. Bixby, Vasek Chvátal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[6] David L. Applegate, William J. Cook, Sanjeeb Dash, and Daniel G. Espinoza. Exact solutions to linear programming problems. *Operations Research Letters*, 35(6):693–699, 2007. `http://dx.doi.org/10.1016/j.orl.2006.12.010`.

[7] David L. Applegate, William J. Cook, Sanjeeb Dash, and Daniel G. Espinoza. QSopt_ex. `http://www.dii.uchile.cl/~daespino/ESolver_doc/main.html`, 2010.

[8] Robert E. Bixby, Sebastián Ceria, Cassandra M. McZeal, and Martin W.P. Savelsbergh. An updated mixed integer programming library: MIPLIB 3.0. *Optima*, 58:12–15, 1998.

[9] William J. Cook, Sanjeeb Dash, Ricardo Fukasawa, and Marcos Goycoolea. Numerically safe Gomory mixed-integer cuts. *INFORMS Journal on Computing*, 21(4):641–649, 2009. `http://dx.doi.org/10.1287/ijoc.1090.0324`.

[10] Sven de Vries and Rakesh Vohra. Combinatorial Auctions: A Survey. *INFORMS Journal on Computing*, 15(3):284–309, 2003.

[11] M. Dhiflaoui, S. Funke, C. Kwappik, K. Mehlhorn, M. Seel, E. Schömer, R. Schulte, and D. Weber. Certifying and repairing solutions to large LPs, how good are LP-solvers? In *SODA 2003: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 255–256, Philadelphia, USA, 2003. Society for Industrial and Applied Mathematics.

[12] Daniel G. Espinoza. *On linear programming, integer programming and cutting planes*. Ph.D. thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology, 2006.

[13] GMP. GNU multiple precision arithmetic library, version 4.3.1. `http://gmplib.org`, 2010.

[14] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[15] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software: Practice and Experience*, 26(8), 1996.

[16] IBM. *CPLEX V12.2, User's Manual for CPLEX*. IBM ILOG, 2010.

[17] Thorsten Koch. The final NETLIB-LP results. *Operations Research Letters*, 32,(2):138–142, 2004.

[18] Carsten Kwappik. *Exact Linear Programming*. Master thesis, Universität des Saarlandes, 1998.

[19] Hans D. Mittelmann. Benchmarks for Optimization Software. `http://plato.asu.edu/bench.html`, 2010.

[20] Arnold Neumaier and Oleg Shcherbina. Safe bounds in linear and mixed-integer linear programming. *Mathematical Programming*, 99(2):283–296, 2004. `http://dx.doi.org/10.1007/s10107-003-0433-3`.

[21] SCIP. Solving constraint integer programs. `http://scip.zib.de`, 2010.

[22] Daniel E. Steffy. *Topics in Exact Precision Mathematical Programming*. Ph.D. thesis, Algorithms, Combinatorics and Optimization, Georgia Institute of Technology, Forthcoming.

[23] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. *SIGPLAN Notices*, 35(5):121–133, 2000.