Math. Prog. Comp. (2013) 5:305–344
DOI 10.1007/s12532-013-0055-6

FULL LENGTH PAPER

A hybrid branch-and-bound approach for exact rational mixed-integer programming

William Cook · Thorsten Koch · Daniel E. Steffy · Kati Wolter

Received: 13 April 2012 / Accepted: 15 May 2013 / Published online: 30 June 2013

Abstract We present an exact rational solver for mixed-integer linear programming that avoids the numerical inaccuracies inherent in the floating-point computations used by existing software. This allows the solver to be used for establishing 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 methods for all binding computations in the search tree. Computing provably accurate solutions by dynamically choosing the fastest of several safe dual bounding methods depending on the structure of the instance, our exact solver is only moderately slower than 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

This Research was supported by NSF Grant CMMI-0726370, ONR Grant N00014-12-1-0030, and the DFG Priority Program 1307 "Algorithm Engineering".

W. Cook
School of Industrial and Systems Engineering, Georgia Institute of Technology,
Atlanta, GA, USA
e-mail: bico@isye.gatech.edu

T. Koch · K. Wolter (✉)
Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
e-mail: wolter@zib.de

T. Koch
e-mail: koch@zib.de

D. E. Steffy
Department of Mathematics and Statistics, Oakland University, Rochester, MI, USA
e-mail: steffy@oakland.edu

the MIPLIB and Mittelmann libraries and for a new collection of numerically difficult instances.

Keywords Mixed integer programming · Branch-and-bound · Exact computation

Mathematics Subject Classification (2000) 90C10 · 90C11 · 90C57

1 Introduction

Mixed-integer programming (MIP) is a powerful and flexible tool for modeling and solving decision problems. Software based on these ideas is utilized 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 (FP) numbers [25]. FP-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, in the methods used for creating cutting planes to improve these relaxations and in pre-solving routines applied to strengthen models.

Due to a number of reasons, for many industrial MIP applications near optimal solutions are sufficient. CPLEX [26], for example, terminates if the relative gap between upper and lower bound is less then 0.001 (relative MIP optimality tolerance). 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.

- Chip design verification in the VLSI design process [2].
- Compiler optimization, including instruction scheduling [45].
- Combinatorial auctions [19], where serious legal and financial consequences can result from incorrect solutions.

Chip design verification and compiler optimization are applications where demonstrating that a particular MIP instance has no feasible solutions is equivalent to verifying the correctness of a proposed point. For pure feasibility problems such as these, accurate answers are extremely important.

The article describing the latest version of the mixed-integer programming library, MIPLIB 2010, discusses the limitations of finite-precision arithmetic in the context of mixed-integer programming [29]. Problem instances were collected from a wide range of applications and a number of the instances were classified as numerically unstable. We now report some computational behavior observed on these instances after they were passed to different solvers using a variety of parameter settings. When called to solve the instance `transportmoment`, under default parameter settings, SCIP 2.1 [2,3,46] (using the SOPLEX 1.6 [47] LP solver) reports to have found an optimal solution, while CPLEX 12.3 claims that the instance is infeasible or unbounded. However, if presolving and cutting planes are disabled, SCIP claims the problem to be

unbounded, (but warns of an error in the proof of unboundedness), and CPLEX reports finite primal and dual bounds. Another example from MIPLIB 2010 is the instance `ns2122603` which at the printing of the paper [29] was incorrectly thought to be infeasible, the answer returned by CPLEX 12.2 (and 12.3); after disabling presolving in CPLEX, a feasible solution can quickly be identified.

Other examples of numerically difficult MIPs occur in the chip design verification instances collected by Tobias Achterberg [2]. There are a total of 98 instances, which are publicly available for download [1]. These instances model property checking on simple arithmetic logical units (ALU). Proving infeasibility of an `alu` instance certifies the correctness of the unit, whereas a feasible solution gives a counter example to the correctness of the design. Although the instances are pure IPs defined by integral data, incorrect conclusions are reached on some of them. For example, the instance `alu10_7`, when calling SCIP 2.1 or CPLEX 12.3 with default settings or with cutting planes and presolving disabled, we get the three different solution values 83, 84, 91. However, none of these values are correct as, by construction, the instance is known to be infeasible. The solutions returned by the solvers only violate the constraints by a small amount and satisfy the relative tolerance thresholds used to measure feasibility, so they are accepted as valid solutions and returned to the user. Further numerically difficult instances are presented in Sect. 6.

Software libraries such as the GNU Multiple Precision Arithmetic Library (GMP) [24] offer routines for infinite-precision rational arithmetic; in contrast to the commonly used finite-precision arithmetic systems, GMP dynamically allocates as much memory as is necessary to exactly represent numbers and is limited only by the available system memory. We use the terms *symbolic* or *exact* when referring to this type of exact computation over the rational numbers; we use the terms *numeric* or *approximate* when referring to the use of inexact finite-precision and floating-point computation. One straightforward strategy to solve MIPs exactly would be to implement the standard solution procedures entirely in exact arithmetic. Unfortunately, it has been observed that optimization software relying exclusively on exact arithmetic can be prohibitively slow [8]. This motivates the development of more sophisticated algorithms 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 [8,20,22,27,30], including the release of the software QSOPT_EX [7]. Exact MIP has seen less computational progress, but significant first steps have been taken. An article by Neumaier and Shcherbina [35] describes methods for safe MIP computation, including strategies for generating safe LP bounds, infeasibility certificates, and cutting planes. Their methods include directed rounding and interval arithmetic with FP-numbers to avoid incorrect results.

This article introduces a hybrid branch-and-bound approach for solving MIPs exactly over the rational numbers. It can be extended to a branch-and-cut algorithm with primal heuristics and presolving; but the focus of this article is on the development of the basic branch-and-bound approach. Section 2 describes how exact rational and safe-FP computation can be coupled together, providing a fast and general framework for exact computation. Section 3 discusses several methods for computing valid LP bounds, a critical component of the hybrid approach. It 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. In Sect. 4, the implementation is further improved by incorporating sophisticated branching rules. The resulting exact solver is compared against a floating-point solver restricted to pure branch-and-bound and observed to be only moderately slower. In Sect. 5, it is used to test the accuracy of current floating-point MIP solvers and in Sect. 6 it is applied to a test set of numerically difficult instances. As our focus has been exclusively on the branch-and-bound procedure, the exact solver is still not directly competitive with the full version of SCIP. However, it is realistic to think that the future inclusion of additional MIP machinery such as cutting planes, presolving, and primal heuristics into this exact framework could lead to a full featured exact MIP solver that is not prohibitively slower than its inexact counterparts.

2 Hybrid rational/safe floating-point approach

Two ideas for exact MIP proposed in the literature, and tested to some extent, are the *pure rational approach* [8] and the *safe-FP approach* [17,35]. 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 rational numbers, and by applying an exact LP solver [22] in the dual bounding step. This approach is especially interesting because it can handle a broad class of problems: MIP instances described by rational data. However, replacing all FP-operations by rational computation increases running times significantly. For example, while the exact LP solver QSOPT_EX avoids many unnecessary rational computations and is efficient on average, Applegate et al. [8] observed a greater slowdown when testing an exact MIP solver that relied on rational arithmetic and called QSOPT_EX for each node LP computation (see also Sect. 3.1).

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, extra work is necessary to ensure correct decisions in the branch-and-bound algorithm. Correctness of certain computations can be ensured by controlling the rounding mode for FP-operations. Valid dual bounds can often be obtained by post-processing approximate LP solutions; this type of safe dual bounding technique has been successfully implemented in CONCORDE [6] for the traveling salesman problem. A generalization of the method for MIPs is described in [35]. Furthermore, the idea of manipulating the rounding mode can be applied to cutting-plane separation. In [17], 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 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 [35] requires, in general, lower and upper bounds on all variables. Weakness in the safely generated bound values may also increase the number of nodes processed by the branch-and-bound solver. Additionally, due to numerical difficulties, some branch-and-bound nodes may only be processable by an exact LP solver.

To summarize, 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 solve MIPs that are given by rational data efficiently and exactly 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 procedures. The *main procedure* implements the rational approach. Its result is surely correct and will be issued to the user. The other one serves as a *slave procedure*, where the faster safe-FP approach is applied. To achieve reasonable running time, whenever possible the expensive rational computation of the main procedure will be skipped and certain decisions from the faster safe-FP procedure will be substituted. In particular, safe dual bound computations in the slave procedure can often replace exact LP solves in the main procedure. The rational procedure provides the exact problem data, allows for the storage of exact primal solutions, and makes exact LP solves possible whenever needed.

The complete procedure is given in Algorithm 1. The set of FP-representable numbers is denoted by $\mathbb{M}$; lower and upper approximations of $x \in \mathbb{Q}$ are denoted $\underline{x} \in \mathbb{M}$ and $\overline{x} \in \mathbb{M}$, respectively. We now explain the details of the algorithm.

The slave procedure, 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 procedures solve the same MIP instance, i.e., $\widetilde{P} := P$ and $\widetilde{c} := c$ in Step 1. 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 the dual bounding method applied in the slave procedure (see Sect. 3).

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 procedures: $\max\{c^T x : x \in LP\}$ with $LP := \{x \in \mathbb{R}^n : Ax \leq b\}$ and $\max\{\widetilde{c}^T x : x \in \widetilde{LP}\}$ with $\widetilde{LP} := \{x \in \mathbb{R}^n : \widetilde{A}x \leq \widetilde{b}\}$. In addition, for each node, we know the branching constraint that was added to create the subproblem in both procedures. Branching on variables, performed in Step 8, introduces the same bounds for both procedures.

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. In particular, in the exact MIP setting, the efficiency highly depends on the strength of the dual bounds and the time spent generating them (Step 5). The starting point of this step is an approximate solution of the LP relaxation of the MIP. It is obtained in the slave procedure by an LP solver that works on FP-numbers and allows rounding errors; referred to as *inexact LP solver*. Depending on the result, we check whether the exact LP relaxation is also infeasible

Algorithm 1 Hybrid branch-and-bound for exact rational MIP

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 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 *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:
      i. If $c^\star > c^{\text{MIP}}$, set $x^{\text{MIP}} := x^\star$, $c^{\text{MIP}} := c^\star$, and $L := \underline{c^\star}$.
      ii. 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.

or we compute a safe dual bound by post-processing the approximate LP solution. Different techniques are discussed in Sect. 3 and are computationally evaluated in Sect. 3.6.

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 test, without tolerances, the integrality of the rational LP solution. Otherwise, we decide if it is worth solving the LP exactly. We deem it worthy if the approximate LP solution is nearly integral. In this case, we solve the LP exactly, using the corresponding basis to warm start the LP solver (hopefully with few pivots and no need to increase the precision) and perform the exact integrality test on the rational 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.

3 Safe dual bound generation

This section describes several methods for computing valid LP dual bounds in Step 5 of Algorithm 1. 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 quickly the bounds can be computed and how strong the bounds are.

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 bound. The fastest exact rational LP solver currently available is QSOPT_EX [7]. The strategy used by this solver can be summarized as follows: the basis returned by a double-precision LP solver is tested for optimality/feasibility by symbolically computing the corresponding basic solution, if it is suboptimal or infeasible then additional simplex pivots are performed with an increased level of precision and this process is repeated until the optimal basis is identified. When possible, the extended precision pivots are warm started with the previously identified LP basis. This method is considerably faster than using rational arithmetic exclusively; it was observed to be only two to five times slower than inexact LP solvers on average over a large test set [8].

In most cases, the double-precision LP run already produced an optimal basis, so the overhead mainly came from computing and verifying the exact rational basic solution. For some instances, this dominates the overall solution time. The costs associated with solving each basis systems exactly may be especially noticeable in the MIP setting. Within a branch-and-bound framework the dual simplex algorithm can be warm started with the final basis computed at the parent node, usually resulting in a small number of dual simplex pivots.

If 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. It is important to note that the solution time of the exact LP solver is influenced not only by the dimension, density, structure, etc., of the LP, but also by the number of bits required to encode the data and solution.

3.2 Basis verification

This strategy avoids the extended precision simplex pivoting that can occur when solving each node LP exactly, but sometimes results in more nodes being processed.

Any exactly feasible dual solution provides a valid dual bound, even if it is not optimal. Therefore, instead of solving each node LP exactly, valid dual bounds can be determined by symbolically computing only the dual solution from a numerically obtained LP basis. If the obtained dual solution is feasible, its objective value gives a valid bound. If it is infeasible, then instead of performing the extra steps required to identify the exact optimal solution, an infinite dual bound is returned. However, if a finite bound was computed at the node's parent, this bound can be inherited, strengthening

an infinite dual bound from basis verification. Within the branch-and-bound algorithm, infinite or weak dual bounds can lead to more branching, but due to the fixing of variables, branching often remediates numerical problems in the LP relaxations down in the tree.

3.3 Primal-bound-shift

Valid bounds can also be produced by correcting approximate dual solutions to be exactly feasible. A special case occurs when all primal variables have finite upper and lower bounds. The following technique was employed by Applegate et al. in the CONCORDE software package [6] and is described more generally for MIPs by Neumaier and Shcherbina [35]. Consider a primal problem of the form $\max\{c^T x : Ax \leq b, 0 \leq x \leq u\}$ with dual $\min\{b^T y + u^T z : A^T y + z \geq c,\ y, z \geq 0\}$. The dual variables z, which correspond to the primal variable bounds, appear as non-negative slack variables in the dual constraints; they can be used to correct any errors existing in an approximate dual solution. 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, but note that this requires the slave procedure to work on an FP-relaxation of the original problem (Step 1 of Algorithm 1).

The simplicity of computing this bound means that it is an excellent choice when applicable. However, if some primal variable bounds are large or missing it may 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 explained below, the most time consuming computations need only be performed once, at the root node of the branch-and-bound tree, and reused for each node bound computation.

To efficiently reuse information through the tree we assume that A^T has full row rank, and that none of the dual variables are implied to be zero. In this case, an LU factorization of a full rank subset of columns S of A^T is computed, this can be reused at every subsequent node of the branch-and-bound tree to compute projections. Also, a point y^* satisfying $A^T y = c,\ y \geq 0$ and $y_i > 0\ \forall i \in S$ is computed at the root node, and will remain dual feasible at all nodes in the branch-and-bound tree. If the root node dual problem is as above, the dual problem at any node can be represented as $\min\{b'^T y + b''^T z : A^T y + A''^T z = c,\ y, z \geq 0\}$ where $b'^T \leq b^T$ and the additional

dual variables z correspond to newly introduced primal variable bounds or cutting planes.

An approximately feasible node dual solution $(\tilde{y}, \tilde{z}) \geq 0$ can be corrected to be exactly feasible by performing the following two steps. First, the violation of the constraints is calculated exactly as $r := c - A^T \tilde{y} - A''^T \tilde{z}$ and a correction vector w satisfying $A^T w = r$ is computed in exact arithmetic using the pre-computed LU factorization; adding w to the approximate solution projects it to satisfy the equality constraints of the problem exactly. This solution $(\tilde{y} + w, \tilde{z})$ might violate the non-negativity constraints, but can only have negative components in the set S. Second, a convex combination of this projected point and y^* is computed as $(\hat{y}, \hat{z}) := (1 - \lambda)(\tilde{y} + w, \tilde{z}) + \lambda(y^*, 0)$, such that $(\hat{y}, \hat{z}) \geq 0$. The resulting point $(\hat{y}, \hat{z})$ is then exactly feasible since it is a convex combination of two points satisfying all of the equality constraints and gives a valid dual bound $b'^T \hat{y} + b''^T \hat{z}$.

Thus, the root node computations involve solving an auxiliary LP exactly to obtain the point y^* 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. This method is more generally applicable than the primal-bound-shift method, but relies on some conditions that are met by most, but not all, of the problems in our test set. Namely, it is assumed that A^T has full row rank and that no dual variables are implied to be zero. A detailed description and computational study of this algorithm can be found in [43]. A related method is also described by Althaus and Dumitriu [5].

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 represents some trade-off between these conflicting characteristics. The *exact LP* method is always applicable and produces the tightest possible bound, 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. The relative performance of these dual bounding methods highly depends on the (sub)problem structure, which may change throughout the tree. Therefore, a strategy that combines and switches between the bounding techniques is the best choice for an exact MIP solver intended to efficiently solve a broad class of problems.

In Sect. 3.6, we will evaluate the performance of each dual bounding method presented here and analyze in what situations which technique works best. In a final step, we then study different strategies to automatically decide how to compute safe dual bounds for a given MIP instance. The central idea of the automatic selection strategy 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.

In the first version ("Auto"), Algorithm 1 decides on the method dynamically in Step 5. At each node primal-bound-shift is applied, and in case it produces an infi-

nite bound one of the other methods is applied. The drawbacks are that it allows for unnecessary computations and that it requires an FP-relaxation for the slave procedure in order to support primal-bound-shift. Alternatively, we can guess whether primal-bound-shift will work ("Auto-Static"). Meaning the dual bounding method is selected depending on the problem structure at the beginning of Algorithm 1, in Step 1, and remains fixed throughout the tree. This allows us to work with FP-approximations whenever we do not select primal-bound-shift. As we will see in the following section, dynamically choosing the dual bounding method at each node achieves superior performance.

After establishing that the dynamic choice of the bounding method is a good strategy, we consider additional ideas, giving two different variants of the "Auto" setting. First, in the "Auto" setting, safe dual bounds are computed at every branch-and-bound node. We will analyze ("Auto-Limited"), whether it is better to compute them only at those nodes where it is required, i.e., if the unsafe dual bound coming from the approximate LP solution would lead to pruning (using tolerances for the comparison with the primal bound). Pruning decisions are critical for the correctness of the final result and have to be safely verified, whereas correct dual bounds are not required for subproblems which will be further processed by branching.

Second, we experiment with interleaving our selection strategy "Auto" with exact LP solves ("Auto-Ileaved"). A safe dual bound obtained by primal-bound-shift can be weaker than the exact LP bound. Sometimes this difference slows down the solution process unnecessarily strong because the solver keeps branching in subtrees that would have been cut off by the exact LP bound. To eliminate some of these cases, we compute the exact LP bound whenever the safe bound from primal-bound-shift is very close to cutting off the node, but not close enough. More precisely, if the bound is within a tolerance smaller than or equal to the primal bound, but not without the tolerance. The hope is that the exact LP bound is slightly stronger and then cuts off the node. Computational results and additional discussion about these ideas are given in Sect. 3.6.3.

3.6 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 at the root node, to study their behavior when applied to a single LP, then examine them within the branch-and-bound algorithm. At the end, we discuss and test strategies to automatically switch between the most promising bounding techniques.

As explained in Sect. 3.3, we have to create an FP-relaxation of the original problem in Step 1 of Algorithm 1 when we want to apply primal-bound-shift, whereas we can use an FP-approximation for the other bounding methods. The discussed algorithms were implemented into the branch-and-bound algorithm provided by the MIP framework SCIP 1.2.0.8 [2,3,46], using best bound search with plunging for node selection and first fractional variable branching. All additional features of SCIP,

like cutting planes, presolving, and primal heuristics, were disabled. For comparison, we also consider the corresponding inexact version of SCIP, i.e., the pure branch-and-bound algorithm with the same node selection strategy and branching rule as in the exact setting ("Inexact"). To solve LPs approximately and exactly we call CPLEX 12.2 [26] and QSOPT_EX 2.5.5 [7], respectively. Rational computations are based on the GMP library 4.3.1 [24]. In the following, we will refer to the above version numbers of the software packages if not otherwise stated. All benchmark runs were conducted on 2.5 GHz Intel Xeon 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 24 h and a memory limit of 13 GB. The timings used to measure computation times are always rounded up to one second if they are smaller. We used the same set-up for the experiments in Sect. 4, 5, and 6.

Our test set contains all instances of the MIPLIB 3.0 [10] and MIPLIB 2003 [4] libraries and from the Mittelmann collections [33] that can be solved within 2 h by the inexact branch-and-bound version of SCIP ("Inexact"). This gives a test suite of 57 MIP instances (see Table 2), which we call *easy test set*. Note that we also analyzed the performance on the remaining, harder, instances of the libraries. The conclusions drawn here, on the smaller suite, were confirmed by these results. At the root node, the individual dual bounding methods were applicable for a similar percentage of instances and also computed bounds of good quality. The overall slowdown factor of the exact solver ("Auto-Ileaved" versus "Inexact") can be expected to be in the same order as for the easy test set. We drew this conclusion by looking at the number of branch-and-bound nodes which both solvers had processed when they hit a certain time limit (but had not solved the instance to optimality). On the easy test set, as we will see later, the exact and the inexact solver require a similar number of branch-and-bound nodes to solve an instance to optimality. This is because the considered benchmarking libraries contain mainly instances that do not cause serious numerical difficulties, and therefore, we expect the number of nodes processed within the time limit to be a good indicator of how much slower the exact code will be.

The easy test set will also be used in Sects. 4 and 5 for studying different branching rules and checking the accuracy of the inexact version of SCIP, respectively. In Sect. 6, we will analyze our exact solver on numerically more difficult instances.

3.6.1 Root node performance

We start by evaluating the root node behavior of the dual bounding methods. Our performance measures are: time overhead and bound quality. The performance profile, see [21], in Fig. 1 visualizes the relative overhead times for the safe dual bounding methods. For each of them, it plots the number of instances for which the safe dual bounding step was performed within a given factor of the bounding time of the fastest method. Table 1 presents the geometric mean of these additional safe dual bounding times in seconds ("DB") and states the number of instances for which a certain dual bound quality was achieved.

This 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

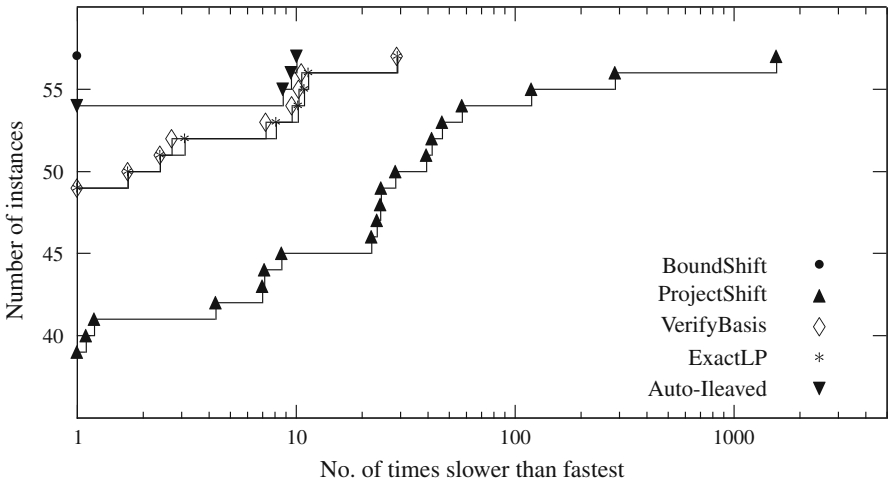

Fig. 1 Comparison of safe dual bounding times "DB" at root node on *easy test set*

Table 1 Safe dual bounding at root node on *easy test set*: relative difference to "ExactLP" dual bound and additional computation time "DB" in geometric mean

Setting	Zero	S	M	L	∞	DB (s)
BoundShift	13	26	2	0	16	1.0
ProjectShift	19	31	5	0	2	2.8
VerifyBasis	57	0	0	0	0	1.3
ExactLP	57	–	–	–	–	1.4
Auto	20	35	2	0	0	1.3
Auto-Static	21	34	2	0	0	1.3
Auto-Ileaved	20	35	2	0	0	1.3

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.

We observe that basis verification has a similar behavior as exact LP for the root node. Still, as we will see in the next section, it gives an improvement over the exact LP solver when expensive basis repair steps are required to find the exact LP solution at certain branch-and-bound nodes.

As expected, primal-bound-shift is the fastest method. However, it produces infinite dual bounds on 16 instances in contrast to only two fails[1] for project-and-shift and

[1] Project-and-shift fails to produce finite bounds on two instances, `swath1` and `swath2`. The conditions necessary for its applicability within the tree (see Sect. 3.4) were not satisfied by these instances.

no fails for basis verification. This is, the bases obtained by CPLEX are often dual feasible and even optimal and project-and-shift meets its requirements most of the time. Still, the finite bounds provided by primal-bound-shift are of very good quality; most of them fall into the "Zero" and "S(mall)" categories. Thus, 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.

Where basis verification is in most cases only up to 10 times slower than primal-bound-shift, project-and-shift is up to 100 times slower at the root node because of the expensive initial set-up step. In the next section, we will see that the overhead incurred by the set-up step of project-and-shift often pays off when it is applied within the entire branch-and-bound tree.

3.6.2 Overall performance

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 ("Inexact"). Table 3 reports the number of instances that were solved within the imposed limits (Column "slv"), for each setting. On 37 instances all settings succeeded. For this group, we present in Table 3, the number of branch-and-bound nodes "Nodes", the solution time "Time" in seconds, and the additional time spent in the safe dual bounding step "DB" in seconds, all in geometric mean for each method. In addition, Fig. 2 gives a performance profile comparing the solution times. For a setting where an instance had a timeout, it is reported with an infinite ratio to the fastest setting. Thus, the intersection points at the right border of the graphic reflect the "slv" column. "Nodes" and "Time" for the individual instances are reported in Table 2. When a dual bounding method leads to a solving time that is within 5 % of the fastest run, the "Time" entry is put in bold. Details for the inexact run can be found in Table 5 ("Inexact-Firstfrac"). Note that in the next section, "Inexact" will be referred to as "Inexact-Firstfrac" in order to emphasis the applied branching rule.

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. However, basis verification is often outperformed by project-and-shift.

Concerning the quality of the safe dual bounds, project-and-shift, basis verification, and exact LP solves perform equally well, which is reflected in a similar (e.g., `bell3a`, `misc07`, and `rentacar`), or even identical (e.g., `acc-0`, `nug08`, and `vpm1`), number of branch-and-bound nodes, see Table 2. Minor node count variations between these exact versions can be explained by slightly different safe dual bounds leading to different node selection decisions. This can, for example, change the point in time when nodes can be cut off due to a new primal solution. It also explains why weaker dual bounds occasionally result in lower overall node counts (e.g., "VerifyBasis" can solve `neos21` using fewer nodes than "ExactLP"). On the instances, where no bounds on the variables are missing, i.e., where primal-bound-shift will always work, the node count is often even similar for all four dual bounding methods. However, the variation is slightly more significant for

Table 2 Overall performance with first fractional variable branching and different safe dual bounding methods on *easy test set*

Example	BoundShift		ProjectShift		VerifyBasis		ExactLP	
	Nodes	Time (s)	Nodes	Time (s)	Nodes	Time (s)	Nodes	Time (s)
30:70:4_5:0_95:100	>948,382	>86400.0	>250,119	>86400.0	190	**47.6**	190	76.9
acc-0	52	**2.5**	52	3.3	52	3.5	52	4.1
acc-1	3,224	**269.3**	3,224	337.1	3,224	393.8	3,224	433.0
acc-2	241	**39.7**	241	45.8	241	51.9	241	56.5
air03	21	**1.1**	21	30.1	21	3.7	21	3.4
air05	94,281	**3795.1**	94,269	15474.3	94,283	14680.2	94,283	19348.7
×bc1	237,946	**22829.1**	>225,758	>86400.0	>75,465	>86400.0	>52,189	>86400.0
×bell3a	>218,056,821	>86400.0	362,609	**564.4**	362,615	3208.2	362,615	3674.9
×bell5	>378,835,894	>86400.0	408,929	**480.0**	408,992	3129.9	408,992	3671.0
×bienst1	96,695	836.2	42,018	**296.0**	40,898	844.8	40,898	811.1
×bienst2	1,137,212	7470.6	447,178	**3768.6**	447,177	8908.3	447,177	8613.2
×blend2	>87,322,668	>86400.0	44,988	**207.0**	44,992	471.1	44,992	583.5
×dano3_3	>11,926	>86400.0	40	**121.2**	40	387.9	40	401.8
×dano3_4	>12,137	>86400.0	193	**415.8**	193	1769.7	193	1864.3
×dano3_5	>13,552	>86400.0	4,722	**8033.9**	4,720	43126.0	4,718	62674.6
×dcmulti	>33,470,968	>86400.0	20,133	**115.6**	20,133	159.7	20,133	215.3
×egout	121,777	**24.4**	60,871	121.5	60,871	331.5	60,871	359.6
eilD76	236,181	**2056.5**	236,305	14369.9	236,305	12219.5	236,303	16786.7
enigma	128,282	**20.9**	128,058	162.7	128,058	449.6	128,058	353.1
×flugpl	4,922	**1.0**	3,519	**1.0**	3,519	1.1	3,519	1.6
×gen	43,265	**38.1**	34,100	397.0	34,100	490.0	34,100	613.4

Table 2 continued

Example	BoundShift		ProjectShift		VerifyBasis		ExactLP	
	Nodes	Time (s)	Nodes	Time (s)	Nodes	Time (s)	Nodes	Time (s)
×gesa3	>19,356,446	>86400.0	128,210	**2645.3**	128,210	3463.2	128,210	4696.8
×gesa3_o	>28,503,643	>86400.0	178,437	**3366.9**	178,437	4408.4	178,437	5860.4
irp	111,775	**6057.0**	116,177	33107.1	116,177	28149.7	116,177	43510.8
×khb05250	6,606	**3.0**	6,606	27.7	6,606	39.5	6,606	43.5
l152lav	11,934	**28.0**	11,933	332.3	11,934	228.0	11,934	279.1
lseu	781,943	**88.2**	795,963	730.1	795,963	717.0	795,963	865.6
×markshare1_1	>207,456,702	>86400.0	>126,148,808	>86400.0	>11,372,254	>86400.0	>10,278,529	>86400.0
×markshare4_0	7,205,565	**575.2**	3,826,128	1551.6	3,826,114	23191.5	3,826,096	22074.2
mas76	7,414,402	**1804.6**	7,568,599	82284.8	>6,281,758	>86400.0	>3,593,826	>86400.0
mas284	1,709,343	**1095.2**	1,894,754	72074.8	1,709,652	34002.6	1,709,652	47618.4
×misc03	1,561	**1.0**	1,561	4.0	1,559	3.9	1,559	4.7
×misc07	368,164	**369.6**	368,179	2550.8	367,676	3127.2	367,676	3564.5
mod008	57,762	**8.9**	59,211	283.7	59,211	443.4	59,211	587.5
mod010	93,730	**200.2**	93,732	2911.3	93,730	1972.8	93,730	2682.4
×mod011	>1,056,416	>86400.0	421,651	67906.5	421,651	**56640.5**	421,651	80443.6
neos5	49,585,878	**13715.0**	>28,412,949	>86400.0	26,371,494	59144.2	26,371,494	63644.7
neos8	25,091	**2228.8**	24,928	8593.2	25,091	68162.0	25,091	72125.8
×neos11	46,712	3331.2	32,006	**2714.3**	30,020	2941.3	30,020	3059.2
×neos21	830,078	**5764.9**	830,716	15662.0	818,609	21096.9	818,611	23479.2
neos897005	86	**339.3**	86	706.1	86	373.0	86	392.8
nug08	143	**14.6**	143	19.0	143	39.0	143	42.3

Table 2 continued

Example	BoundShift		ProjectShift		VerifyBasis		ExactLP	
	Nodes	Time (s)	Nodes	Time (s)	Nodes	Time (s)	Nodes	Time (s)
nw04	10,826	**1402.4**	10,826	16235.5	10,826	12127.2	10,826	12097.7
p0033	2,664	**1.0**	2,670	**1.0**	2,670	1.1	2,670	1.3
p0201	5,746	**2.8**	5,788	19.5	5,780	16.1	5,780	30.5
×pk1	>425,303,108	>86400.0	1,793,664	**6509.8**	1,793,663	19795.5	1,793,656	21516.5
qap10	246	**548.9**	246	**573.9**	246	979.0	246	2120.3
×qnet1_o	981,487	**1421.9**	730,464	12195.0	731,031	16823.3	731,031	19706.4
×ran13x13	>383,471,711	>86400.0	>26,422,361	>86400.0	>28,943,888	>86400.0	>27,372,116	>86400.0
×rentacar	321,915	20848.9	165	70.3	156	**31.0**	156	47.3
rgn	10,206	**9.5**	10,249	28.9	10,249	94.2	10,219	145.2
stein27	4,031	**1.0**	4,031	2.8	4,031	4.4	4,031	4.9
stein45	58,333	**31.2**	58,329	97.4	58,333	167.0	58,333	182.0
×swath1	1,890,605	48739.2	>1,677,398	>86400.0	560,996	**44176.7**	560,996	69978.7
×swath2	2,707,605	**65573.1**	>1,664,965	>86400.0	>1,099,065	>86400.0	>716,964	>86400.0
vpm1	7,773,158	**1962.9**	7,773,158	25645.6	7,773,158	18726.1	7,773,158	20405.2
vpm2	27,383,880	**8652.6**	>21,823,235	>86400.0	>19,267,535	>86400.0	>17,032,855	>86400.0

Detailed results. Instances missing bounds on variables are marked by "×" . Solving times within 5% of the fastest setting are put in bold

Table 3 Summary of overall performance with first fractional variable branching on *easy test set*

Setting	slv	Geometric mean for instances solved by all settings (37)		
		Nodes	Time (s)	DB (s)
Inexact	57	18,030	59.4	–
BoundShift	43	24,994	110.4	13.9
ProjectShift	49	18,206	369.3	238.1
VerifyBasis	51	18,078	461.8	329.8
ExactLP	51	18,076	550.7	419.0
Auto	54	18,276	92.6	17.5
Auto-Static	53	18,276	100.2	19.8
Auto-Ileaved	55	18,226	91.4	18.4
Auto-Limited	48	22,035	89.9	12.0

"slv" number of instances solved, "DB" safe dual bounding time

primal-bound-shift, because an FP-relaxation of the original problem is set up in Step 1 of Algorithm 1 instead of an FP-approximation; relative to the others, this may produce different approximate LP solutions. Sometimes this even leads to fewer nodes for primal-bound-shift (e.g., `rgn`). Table 2 also gives an example (`khb05250`) for an instance where primal-bound-shift works even though bounds are missing on some variables; these bounds were not required in the correction step.

Concerning the overall solution time, we observe that, when applicable, primal-bound-shift computes valid dual bounds with very little overhead. For the instances it solved we usually experience a slow-down of at most 2, relative to the inexact branch-and-bound solver. The few large slow-down factors of up to 10 can be explained by a node increase due to a small number of missing variable bounds and by expensive exact LP calls for computing primal bounds. The one extreme slow-down factor comes from `rentacar`, which is solved by pure enumeration; primal-bound-shift produces infinite bounds at all nodes. However, due to its limited applicability it solved only 43 instances within the imposed limits.

In contrast, project-and-shift solves 49 instances. The dual bound quality was strong enough such that instances could be solved without requiring a significant increase in the number of nodes processed, relative to the "ExactLP" strategy. In the previous section we observed a large overhead required at the root node by this method, making it impractical for computing valid bounds on a single LP; however, we observe that amortized over the entire branch-and-bound tree, the resulting solution time is competitive. In mean, it is only 6 times slower than the inexact code. In this fashion, most of the instances were solved within 20 times the time used by the inexact code.

If we compare project-and-shift with basis verification (Table 2; Fig. 2) we see a similar and often better performance for project-and-shift. Still, on some instances basis verification works better. For example, it solves two more instances of our test set. To figure out when we should choose basis verification instead of project-and-shift, i.e., when (the setup step of) project-and-shift is too expensive, we tested different, easy to compute, problem characteristics. Since the setup costs of project-

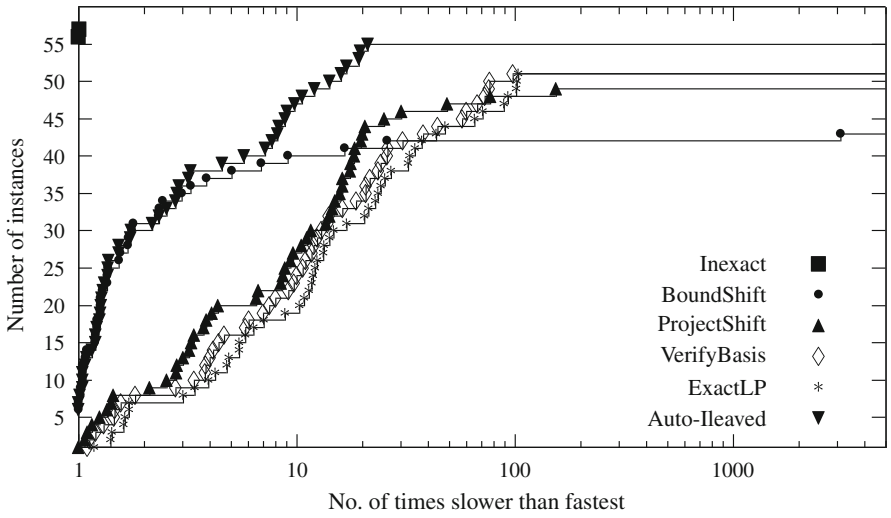

Fig. 2 Comparison of overall solving times "Time" on *easy test set*. Branching rule is first fractional variable branching

and-shift are dominated by an exact LP solve and a symbolic LU factorization (see Sect. 3.4), we tested criteria in connection with the constraint matrix: dimension of the matrix, ratio between number of rows and columns, percentage of nearly parallel rows (sometimes introduced to build an FP-relaxation), number of non-zeros, percentage of integral non-zeros, ratio between largest and smallest absolute value. Another idea was to estimate whether project-and-shift will be called often enough such that the setup step pays off. Here, we looked at the percentage of variables with missing bounds. The best results were obtained with the number of non-zeros in the constraint matrix. In the automatic dual bound selection strategies, discussed below, we prefer project-and-shift as long as the matrix has at most 10,000 non-zeros.

Only one instance, `markshare1_1`, could not be solved within the imposed limits by any of the four exact versions. In contrast to the other instances, the node count for `markshare1_1` significantly increases with the exact solvers, see Tables 2 and 5. The reason is that in the course of the branch-and-bound processes some of the nearly integral approximate LP solutions do not correspond to integral exact LP solutions, which causes many additional branchings; in particular, this holds for the final primal solution found by the inexact solver.

3.6.3 Combinations

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: on our test set, running project-and-shift on an FP-relaxation gave similar results to running it on an FP-approximation. Therefore, we decided to always set up an FP-relaxation

in Step 1 of Algorithm 1. This way, we are allowed to apply primal-bound-shift at any node 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 project-and-shift or basis verification depending on the constraint matrix as explained above. The root node results for the combined versions are presented in Table 1 and Fig. 1; the overall performance results can be found in Table 3 and Fig. 2. Note that we excluded "Auto-Limited" from Table 1 as it never computed safe finite bounds at the root node and that we only included the best auto setting in the performance profiles as their graphs look very similar. Detailed results for the inexact run and the best auto setting are given in Table 5. Notice that in this table for reasons of clarity, "Inexact" and "Auto-Ileaved" are called "Inexact-Firstfrac" and "Exact-Firstfrac", respectively.

The experiments show that "Auto" combines the advantages of all dual bounding methods. We can solve all 43 instances that primal-bound-shift solved as well as 11 additional ones by automatically switching to other dual bounding methods at the nodes. In Sect. 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 the 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 problem 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 at each node whether primal-bound-shift works, and it even leads to a slightly improved performance.

The second idea was to interleave the strategy with exact LP calls ("Auto-Ileaved"). This strategy tries to avoid situations when branching is applied repeatedly on nodes that could be safely cut off if their LPs were solved exactly, but not if a weaker bound was computed. Examples are nodes where the exact LP dual bound is equal to the best known primal bound. This situation does not occur on many instances in our test set, but when it does, the interleaving strategy is helpful. We solve one more instance (`30:70:4_5:0_95:100`) to optimality without introducing a significant time overhead on the other instances.

The third extension was to only compute bounds safely at nodes where the (unsafe) bound coming from the approximate dual solution would lead to cutting off the 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. It solved fewer instances than the other automatic settings and processed more nodes. On harder instances the node count at timeout was higher than the other methods, i.e., the node processing is much faster on average. However, we cannot draw strong conclusions about the quality of this approach on harder instances, as in this setting the exact primal-dual-gap does not improve steadily. Moreover, one advantage of computing safe dual bounds at more nodes of the branch-and-bound tree is that these safe bounds are inherited by a node's children. Therefore, if safe bounds were computed previously, the discovery of an improved primal bound may allow immediate pruning of many unprocessed nodes. In a similar situation, the setting "Auto-Limited" may incur extra cost computing safe bounds at each of these nodes individually.

4 Branching rules

So far we have introduced a branch-and-bound algorithm for solving MIPs exactly and developed an advanced strategy for computing the dual bounds safely in this framework ("Auto-Ileaved", which we refer to as "Exact-Firstfrac" in the following). Here we will improve the branching step of the current implementation. Choosing which variable to branch on is crucial for MIP solvers. The experiments in [2] (using SCIP version 0.90f) showed that replacing the default branching rule in SCIP by other less sophisticated ones increases the running time by a factor of up to 4. For comparison, disabling cutting plane separation doubles the solution time.

The inexact version of SCIP supports various branching rules. We tested the following ones in the exact MIP setting, listing them in increasing order of their performance in the inexact full version of SCIP as evaluated by Achterberg [2].

- "Exact-Leastinf": Least infeasible branching
- "Exact-Firstfrac": First fractional branching
- "Exact-Mostinf": Most infeasible branching
- "Exact-Fullstrong": Full strong branching
- "Exact-Pseudocost": Pseudocost branching
- "Exact-Reliability": Reliability branching

Least infeasible and *most infeasible branching* consider the fractional parts of the integer variables in the LP solution. By solving the LP relaxation of the potential subproblems for all branching candidates, *full strong branching* chooses the variable which leads to the best dual bound improvement. *Pseudocost branching* tries to estimate this improvement by analyzing the dual bound gains achieved by previous branchings. *Reliability branching* uses strong branching only on variables with unreliable pseudocosts, i.e., with a limited branching history. *First fractional branching*, the rule used so far in our implementation, simply decides to branch on the first integer variable (w.r.t. variable index) with fractional LP solution value. SCIP applies this scheme when no special branching rule is implemented. It was not tested in [2], but the performance is in the range of most infeasible and least infeasible branching.

When selecting the branching variable in the exact MIP setting, exact branching score calculation is not required to obtain a correct solution. In particular, the strong branching LPs do not need to be solved exactly. The only restriction is that all other conclusions drawn from strong branching LPs are ignored; they are not safe anymore if the LPs are only solved by an inexact LP solver. These additional conclusions include prunable subproblem detection (if we find a branching candidate for which both strong branching LPs are infeasible), domain reduction (of all variables for which one of the strong branching LPs is infeasible), and dual bound improvement (the weaker objective function value of the two strong branching LP solutions of a variable provides a valid dual bound for the current subtree).

For full strong branching, this significantly reduces its potential. Table 4 summarizes, alongside others, the results for running the inexact branch-and-bound algorithm of SCIP without additional conclusions from strong branching LP solutions ("Inexact-Fullstrong") and with them ("Inexact-Fullstrong+"). Supporting this step, as done in standard floating-point MIP solvers, speeds up the branch-and-bound process by a

Table 4 Summary of performance for different branching rules on *easy test set*

Setting	slv	Geometric mean for instances solved by all settings (47)		
		Nodes	Time (s)	DB (s)
Exact-Leastinf	49	48,211	324.3	84.7
Exact-Firstfrac	55	27,306	193.2	56.7
Exact-Fullstrong	55	3,146	168.2	12.9
Exact-Mostinf	54	11,935	98.4	34.5
Exact-Pseudocost	56	6,745	53.4	20.0
Exact-Reliability	56	3,394	43.8	13.0
Inexact-Firstfrac	57	26,686	77.0	–
Inexact-Reliability	57	3,458	19.8	–
Inexact-Reliability+	57	2,611	21.5	–
Inexact-Fullstrong	57	2,941	104.6	–
Inexact-Fullstrong+	57	789	58.6	–

Exact solver uses "Auto-Ileaved" for safe dual bounding
"slv" number of instances solved, "DB" safe dual bounding time

factor of 1.8. The node count is reduced by a factor of 3.7. So the positive impact of full strong branching is not only due to good branching decisions based on additional LP solves; to a certain extent it is also achieved by drawing further conclusions from these strong branching LPs, which includes variable fixings, domain reductions and node cutoffs. Regarding the node count improvement, one should keep in mind that if full strong branching "creates" subproblems and detects them to be prunable, they are not counted as branch-and-bound nodes. Performing the same experiment with reliability branching, i.e., if strong branching is only used in case of unreliable pseudocosts, we observe that the additional conclusions have only a very small impact ("Inexact-Reliablity" versus "Inexact-Reliablity+" in Table 4).

For the exact MIP setting, the impact of each tested branching rule is summarized in Table 4. The ranking is similar to what was experienced for the floating-point version of SCIP in [2]; except for full strong branching which performs in our tests worse than most infeasible branching for the reasons explained above. The best results were obtained with reliability branching.

Tables 4 and 5 compare the performance of first fractional branching and reliability branching in the inexact branch-and-bound version of SCIP ("Inexact-Firstfrac" versus "Inexact-Reliablity") and in the exact version ("Exact-Firstfrac" versus "Exact-Reliability"). In both settings, the impact of reliability branching is of the same range. In mean, the running time with this rule improves by a factor of 3.9 for the inexact and 4.4 for the exact code. In addition, Fig. 3 visualizes the changes in running time and in the number of branch-and-bound nodes between the inexact and the exact code when both apply reliability branching. The performance degradation is similar to what was experienced with first fractional branching in the previous section (see Fig. 2; Table 3). In geometric mean, the exact version is only 2.2 times slower than the inexact one. However, with reliability branching the branch-and-bound tree diverges

Table 5 Overall performance of exact and inexact solver with first fractional and reliability branching on *easy test set*

Example	Inexact-Firstfrac		Exact-Firstfrac		Inexact-Reliability		Exact-Reliability	
	Nodes	Time (s)	Nodes	Time (s)	Nodes	Time (s)	Nodes	Time (s)
30:70:4_5:0_95:100	190	**14.0**	379	101.4	155	75.7	309	153.1
acc-0	52	**2.5**	52	**2.5**	56	14.1	56	14.2
acc-1	3,224	268.7	3, 224	269.5	51	**30.4**	51	**30.5**
acc-2	241	**39.6**	241	**39.6**	79	41.7	79	41.8
air03	21	**1.0**	21	1.1	16	1.6	16	2.7
air05	94,281	3537.6	94,281	3792.4	977	60.3	653	**45.7**
×bc1	237,946	5923.5	237,946	18875.4	135,376	**3980.8**	134,702	13895.1
×bell3a	362,615	84.5	354,952	485.7	59,610	**13.0**	56,305	80.6
×bell5	408,992	41.2	417,113	373.8	76,897	**9.2**	413,159	377.8
×bienst1	40,904	91.9	41,103	233.9	20,426	**45.6**	12,214	64.2
×bienst2	445,383	1481.1	447,178	3218.2	94,669	**220.6**	96,100	682.0
×blend2	44,589	8.1	45,305	160.5	8,425	**2.1**	8,719	29.0
×dano3_3	40	**32.2**	40	390.3	33	75.7	33	395.5
×dano3_4	193	126.0	193	1784.9	55	**106.9**	52	611.9
×dano3_5	4,718	2663.2	4,732	42552.8	223	**215.7**	231	2331.0
×dcmulti	20,133	8.5	20,133	89.7	2,142	**1.3**	2,224	11.5
×egout	60,871	7.4	60,871	57.6	8,337	**1.1**	8,832	7.4
eilD76	236,305	1629.2	236,181	2065.5	12,884	**149.0**	15,087	216.3
enigma	128,058	17.3	128,282	21.6	465	**1.0**	465	**1.0**
×flugpl	3,519	**1.0**	3,519	**1.0**	4,203	**1.0**	4,302	**1.0**
×gen	34,100	22.5	34,649	73.6	549	**1.0**	770	5.9
×gesa3	128,210	143.0	128,210	2753.0	6,000	**7.4**	6,049	120.1
×gesa3_o	178,437	168.4	178,437	3562.8	5,705	**7.1**	5,999	111.2

Table 5 continued

Example	Inexact-Firstfrac		Exact-Firstfrac		Inexact-Reliability		Exact-Reliability	
	Nodes	Time (s)	Nodes	Time (s)	Nodes	Time (s)	Nodes	Time (s)
irp	116,177	2035.1	111,775	5886.8	136,272	**1888.1**	11,758	2174.3
×khb05250	6,606	1.9	6,606	2.8	2,816	**1.0**	2,816	1.3
l152lav	11,934	23.4	11,934	28.0	921	**5.5**	1,140	6.6
lseu	781,943	65.3	781,943	88.7	34,937	**3.2**	34,937	4.4
×markshare1_1	2,404,813	**160.0**	>305,281,790	>86400.0	2,282,316	**157.9**	>276,279,298	>86400.0
×markshare4_0	3,826,122	236.9	3,826,122	662.2	1,314,927	**84.5**	1,297,895	239.0
mas76	7,415,279	1067.0	7,414,402	1845.9	1,012,655	155.3	593,334	**140.5**
mas284	1,709,652	466.5	1,709,343	1096.0	30,378	**11.2**	29,892	22.3
×misc03	1,559	**1.0**	1, 559	**1.0**	810	**1.0**	788	**1.0**
×misc07	367,676	293.9	367,676	375.9	27,222	**25.5**	33,885	40.1
mod008	57,768	5.9	57,762	8.8	14,743	**2.0**	14,743	3.0
mod010	93,730	158.9	93,730	200.2	628	**4.2**	627	4.7
×mod011	421,651	7060.8	421,653	62387.6	45,586	**796.7**	52,061	9021.8
neos5	26,371,297	5899.1	26,373,009	16627.7	2,261,391	**519.9**	5,527,463	3770.4
neos8	25,095	2228.8	25,091	2567.9	1,374	**371.6**	4,866	977.0
×neos11	30,020	1875.8	30,034	2460.7	15,035	1271.6	10,785	**1167.5**
×neos21	818,609	5528.5	818,613	5774.6	10,458	**133.1**	9,810	**132.9**
neos897005	86	**334.9**	86	**333.1**	11	359.5	11	350.9
nug08	143	**13.9**	143	14.6	11	23.4	11	23.4
nw04	10,826	1031.7	10,826	1411.5	766	**213.6**	766	260.2
p0033	2,664	**1.0**	2,664	**1.0**	1,388	**1.0**	1,388	**1.0**
p0201	5,746	2.3	5,746	2.8	379	**1.0**	379	**1.0**
×pk1	1,793,663	329.8	1,793,663	5577.8	609,529	**126.5**	609,529	1965.8

Table 5 continued

Example	Inexact-Firstfrac		Exact-Firstfrac		Inexact-Reliability		Exact-Reliability	
	Nodes	Time (s)	*Nodes*	Time (s)	Nodes	Time (s)	Nodes	Time (s)
qap10	246	538.8	246	548.1	13	**143.2**	13	**143.8**
×qnet1_o	731,031	812.2	981,086	1419.5	1,715	**3.9**	1,467	5.0
×ran13x13	27,604,305	5188.0	>26,208,179	>86400.0	795,936	**146.7**	795,961	2538.3
×rentacar	155	**6.6**	169	30.5	61	8.9	61	19.4
rgn	9,877	1.3	10,202	11.5	4,019	**1.0**	5,03	12.5
stein27	4,031	**1.0**	4,031	**1.0**	4,345	**1.0**	4,345	**1.0**
stein45	58,333	28.6	58,333	31.2	54,972	**27.2**	54,961	29.9
×swath1	560,996	1874.1	560,996	18367.0	17,696	**70.0**	17,694	572.0
×swath2	1,156,144	3955.1	1,146,891	32248.0	20,789	**82.4**	20,324	497.7
vpm1	7,773,158	1446.2	7,773,158	1983.7	324,354	**60.5**	341,834	86.4
vpm2	27,384,657	6736.8	27,383,880	8753.5	955,054	**234.7**	959,465	307.8

Detailed results. Exact solver uses "Auto-Ileaved" for safe dual bounding. Instances missing bounds on variables are marked by "×" . Solving times within 5% of the fastest setting are put in bold

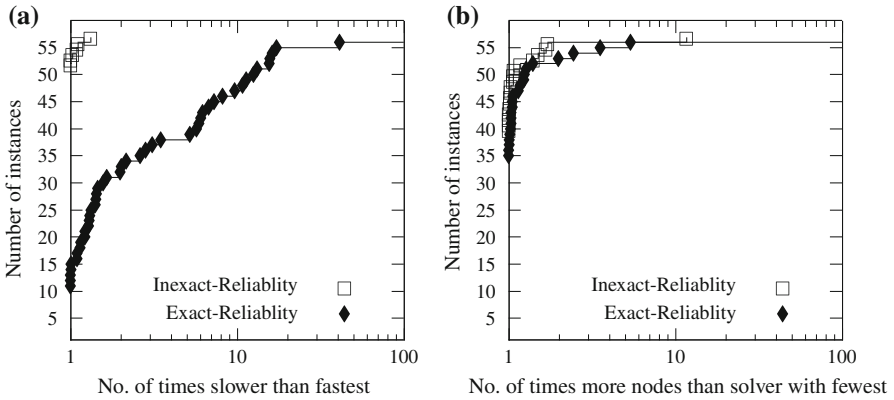

Fig. 3 Comparison of best exact solver and inexact counterpart on *easy test set*. **a** Performance profile for solving time "Time". **b** Performance profile for branch-and-bound nodes "Nodes"

more between the inexact and the exact solver; in Table 5 we observe that the node count differs more with reliability branching than with first fractional branching. In the extreme, both solvers need the same number of branch-and-bound nodes with first fractional branching, while having different node counts with reliability branching (e.g., `air05`, `egout`, and `vpm1`). The reason is that the sophisticated strategy of reliability branching is more sensitive to small changes, for example, in the dual bounds and the number of LP iterations (see [2] for details on the reliability branching algorithm). To summarize, the exact code benefits from better branching rules in the same way as the inexact one.

In addition to standard branching strategies, one that aims at making the fast safe dual bounding method primal-bound-shift (see Sect. 3.3) work more often would be interesting. If a missing bound constraint is necessary to repair the approximate dual solution by primal-bound-shift, the method will fail and we apply one of the more expensive dual bounding methods at this branch-and-bound node. Branching on such variables would introduce a missing bound in one of the created subproblems and this way could increase the chance of primal-bound-shift to be applicable in this subtree. On the easy test set, 29 of the 57 instances contain variables with infinite lower or upper bounds. They are marked by a "×" in Table 5. However, examining the problem characteristics we noticed that all missing bounds were on continuous variables only. That is, we are not able to introduce the required bounds through branching decisions; branching is only performed on integer variables. On numerically difficult instances, considered in the next section, we observed a similar situation. In Table 11, 28 out of 50 instances had infinite bounds, but only in a few cases this involved integer variables (`dfn6_load`, `dfn6fp_load`, `dfn6f_cost`, and `dfn6fp_cost`).

5 How accurate are current MIP solvers?

On the easy test set, with reliability branching, we are able to solve all but one instances exactly (`markshare1_1`). Thus, having exact objective function values for nearly all instances at hand, we now want to analyze how accurate the floating-point version

of SCIP is. In the inexact setting, errors in the branch-and-bound process can be introduced at several different places: while reading in the instance, in the bounding step and in the feasibility test (because of the FP-arithmetic and the consequent usage of tolerances), and because of inaccurate LP solutions. See [29,42] for further discussion regarding possible sources and types of errors that might be encountered.

We considered our best exact branch-and-bound version ("Exact-Reliability") and its inexact counterpart ("Inexact-Reliablity") and present in Table 6 the "Difference" between the objective function values returned by the exact and the inexact run ("Exact Objval" [2] and "Approx Objval").

We mark cases where an instance was not solved to optimality within the limits (see Table 5) by a "—" and also use "—" in the "Difference" column then. Otherwise, the exact absolute difference is computed. If it is non-zero, the closest FP-number is displayed.

For the majority of the instances, the objective values are identical. On 12 instances, the inexact branch-and-bound solver reports results that differ from the exact objective values, but the differences are not significant. This indicates that no dramatic mistakes were made by the FP branch-and-bound solver. But this is not surprising as the instances come from standard MIP libraries, for which numerical troubles are very seldom.

Only `markshare1_1`, which we were not able to solve, is numerically less stable. As explained in Sect. 3.6.2, in contrast to the other instances, the node count for `markshare1_1` significantly increased with the exact solver. The reason is that in the course of the branch-and-bound process some of the nearly integral approximate LP solutions do not correspond to integral exact LP solutions (best primal bound found within the imposed limits is 235/398953428.), which causes additional branchings. On all other easy instances, this did not happen.

Notice that this experiment also shows that all studies on the easy test set were fair. We did not compare solution times for instances where the inexact code terminates quickly, but computes a result that is far from correct. The picture is more diverse on numerically more difficult instances as considered in the next section.

6 Numerically difficult MIP instances

In the last section, we showed that the exact branch-and-bound code was able to solve the problems in our easy test set within a reasonable factor of the time required by the inexact branch-and-bound solver. Here we will analyze its behavior on numerically difficult instances.

6.1 Selecting the test set

Before going any further we must ask: what does it mean for a MIP to be *numerically difficult*? It would be nice if there were some clear, well defined properties that

[2] Of course, even with a very carful implementation and extensive testing, a certain risk of an implementation error remains (also in the underlying exact LP solver and the software package for rational arithmetic). So, the exact objective values reported here come with no warranty.

Table 6 Comparison of exact and approximate objective function values on *easy test set*

Example	Exact-Reliablity Exact Objval	Inexact-Reliablity Approx Objval	Difference
30:70:4_5:0_95:100	3	3.00000000000000e+00	
acc-0	0	0.00000000000000e+00	
acc-1	0	0.00000000000000e+00	
acc-2	0	0.00000000000000e+00	
air03	340160	3.40160000000000e+05	
air05	26374	2.63740000000000e+04	
bc1	5097791055616760409505	3.33836254764631e+00	5.45491510972293e-12
	741025156759544918657977282733349001870022641698621508074764296543670423384251433310667545332570 57/ 152703338324330694722974844929002695674318892373641886227384705977426250447047855421387906252680844296971834035564360269		
bell3a	219607579/250	8.78430316000000e+05	
bell5	28020020286/3125	8.96640649152000e+06	
bienst1	187/4	4.67500000000000e+01	
bienst2	273/5	5.46000000000000e+01	
blend2	1519797/200000	7.59898500000000e+00	
dano3_3	746062958774188756	5.76344633030206e+02	4.37618286123444e-10
	39563005360272004148626307217961730740455687448763160640101226321601863734721821047 401723482988898100746851847053919272535197083424547986697987345747243316460015749 1/ 129447368122628357853850055681478138577407775147917745512634540745794805067059713981042752137417872821240440106741751055704244192505693873724717218993448484095497950942713261286110000		
dano3_4	67206255269530220	5.76435224707201e+02	2.69535256699552e-12
	77397013218912966236130303895160753522695940755911414845904055135781966795786095437838835954130036 28695293315034375753012466846438351173000738585896913211411902479197183558713984871857895781939173/ 11658943171570955876486228925173492000660419263416336006298956645009511892635353747701145341091016 03637641399852024834140764994428695930329822345188426990860299552797117998407603713373282618672734 72511761250000		
dano3_5	1743173141933503	5.76924915956559e+02	2.75504852651767e-13
	23944289922574375643749932000407736302001575501699804011697573580822524418821770737899385131876111 51429598921349829858998392124701737441973395730485310049941446001305541636197399048288165055376 23/ 30214904812063284849917509108363229121498170037716519591132122206836805081257536132318028700350141 64361460271176698537302130592201747497379591768297328768660696668326008580144081308486528857679334 267391476000		
dcmulti	188182	1.88182000000000e+05	
egout	5681007/10000	5.68100700000000e+02	
eilD76	885411847/1000000	8.85411846999999e+02	1.00000000000000e-12
enigma	0	0.00000000000000e+00	
flugpl	1201500	1.20150000000000e+06	
gen	56156681359/500000	1.12313362718000e+05	
gesa3	125881983070952346961091799641922753/ 44972237959212201700000000000	2.79910426483827e+07	1.39669000058314e-08
gesa3_o	125881983070952346961091799641922753/ 44972237959212201700000000000	2.79910426483827e+07	1.39669000058314e-08
irp	3039873209/250000	1.21594928360000e+04	
khb05250	106940226	1.06940226000000e+08	
l152lav	4722	4.72200000000000e+03	
lseu	1120	1.12000000000000e+03	
markshare1_1	—	0.00000000000000e+00	—
markshare4_0	1	9.99999999999993e-01	7.00000000000000e-15
mas76	20002527071/500000	4.00050541420000e+04	
mas284	457028618411/5000000	9.14057236822000e+04	
misc03	3360	3.36000000000000e+03	
misc07	2810	2.81000000000000e+03	
mod008	307	3.07000000000000e+02	
mod010	6548	6.54800000000001e+03	1.00000000000000e-11
mod011	-2101835	-5.45585350142273e+07	3.29093662039378e-08
	06086976561180670715705770017207889097120559747472357930811351596541825 7/ 385244042993761853521994950647178332080944978376263342522033000000000		
neos5	15	1.50000000000000e+01	
neos8	-3719	-3.71900000000000e+03	
neos11	9	9.00000000000000e+00	
neos21	7	7.00000000000000e+00	
neos897005	14	1.40000000000000e+01	

Table 6 continued

Example	Exact-Reliablity Exact Objval	Inexact-Reliablity Approx Objval	Difference
nug08	214	2.14000000000000e+02	
nw04	16862	1.68620000000000e+04	
p0033	3089	3.08900000000000e+03	
p0201	7615	7.61500000000000e+03	
pk1	11	1.10000000000000e+01	
qap10	340	3.39999999999999e+02	1.00000000000000e-12
qnet1_o	16029692681/1000000	1.60296926810000e+04	
ran13x13	3252	3.25200000000000e+03	
rentacar	6130241041408706422121 9/2019398922240000	3.03567609841487e+07	5.06500607490391e-08
rgn	2054999981/25000000	8.21999992400000e+01	
stein27	18	1.80000000000000e+01	
stein45	30	3.00000000000000e+01	
swath1	1516285183/4000000	3.79071295750000e+02	
swath2	7703993859/20000000	3.85199692950000e+02	
vpm1	20	2.00000000000000e+01	
vpm2	55/4	1.37500000000000e+01	

If absolute difference (computed exactly) between "Exact Objval" and "Approx Objval" is non-zero, closest FP number is displayed in "Difference"

would predict which instances could be solved easily using floating-point computation, and which instances would succumb to numerical issues in the solution process. Unfortunately, this question does not have a simple answer.

We first focus our attention to linear programming where a number of authors have studied the related idea of condition measures [13,16,36,39,40]. LPs are considered *ill-conditioned* if small modifications in the problem data can have a large effect on the solution; in particular, if they lead to changes in the optimal objective value, changes in primal or dual feasibility, or changes in the structure of the final LP basis. Connections have been made between LP condition measures and the complexity of solving them [14,15,41,44]; ill-conditioned LPs may require higher precision arithmetic or more interior point iterations. Computational studies have also investigated these ideas [12, 37]. However, LP condition numbers are not always a good predictor that LP instances will or will not be solvable by floating-point software packages. For example, in [37], 71 % of the NETLIB LP instances [9,23] were observed to have infinite condition measures, 19 % after pre-processing. However, in [27], double-precision LP solvers were used to identify the optimal basis for all NETLIB LP instances; this could be seen as an indication that, in some practical sense, these instances are not numerically difficult. Conversely, one could easily construct well conditioned LPs that are unsolvable by double-precision based software by, e.g., scaling the data to contain entries too large or small to be represented by a double-precision number.

Turning our attention back to MIPs, to the best of our knowledge no study has defined or discussed the notion of a condition measure. When switching from continuous to discrete variables arbitrarily small changes in the data defining an instance is more likely to alter the feasibility or optimality. As the nature of our study is computational we will prefer a test set that is numerically difficult in the practical sense – meaning it is composed of instances on which software packages available today compute incorrect or conflicting results or exhibit evidence of incorrect computations within the solution process.

Starting from a total of over 600 instances taken from the unstable test set of the MIPLIB 2010 library [29], the COR@L MIP collection [32,31], instances that were submitted to the NEOS server [18,34] to be solved exactly, and instances from projects at ZIB, we collected a test suite of 50 instances, which we will call the *numerically difficult test set*. Table 7 states the origin and a short description of the chosen instances. Furthermore, Table 8 shows the statistics that were relevant for the selection of these instances; met criteria are put in bold.

We now describe the empirically motivated criteria which we have used to classify instances as numerically difficult. They are based directly on the behavior of floating-point MIP solvers applied to the instances.

One attempt to identify numerical issues during the solving process was recently, with version 12.2, introduced in CPLEX. It considers the condition number of the optimal LP bases at the branch-and-bound nodes and classifies them as *stable*, *suspicious*, *unstable*, and *ill-posed* (see [29] for more details). Even though this measure is highly dependent on the solution process and may not help to identify numerically unstable instances, we found it reasonable to take it as one criterion for the selection of our test set. In Table 8, the first block of columns, "p_{susp}", "p_{unstab}", and "p_{illpo}", states the relative frequency of the sampled bad condition number categories. We used `set mip strategy kappastats 2`, i.e., computed LP condition numbers for every subproblem. Furthermore, CPLEX weights these three groups into one estimate for the probability of numerical difficulties. It is called *attention level* (column "AL"). Since the estimate depends on the solution process, we run the solver with five different parameter settings: default settings, presolving disabled, cuts disabled, primal heuristics disabled, and all three components disabled. The statistics in Table 8 refer to the worst (largest) values observed among the runs (time limit of 2 h), and we display only non-zero values.

Our second indicator of numerical issues is, whether the input data contain values of very different magnitude. The columns "r_{coef}", "r_{rhs}", and "r_{obj}" state the ratio between the largest and the smallest absolute non-zero entry in the coefficient matrix, the right hand side vector, and the objective function vector, respectively. Largest and smallest values are taken from the log files of CPLEX.

As a third point, we checked for inconsistent results returned by different MIP solvers on various parameter settings. We run SCIP 2.0.2 and CPLEX (with `mipkappa` computation, and without) and in both solvers, applied the five settings mentioned above. Columns "db" and "pb" report the maximum dual bound and the minimum primal bound returned at termination among all runs. Notice that all instances have minimization form. In case of infeasibility detection, we work with primal and dual bounds of 10^{20} and display "lnfeas" in Table 8. We selected instances that meet one of the following criteria

- Unstable LPs: "AL" ≥ 0.1, "p_{susp}" ≥ 0.5, "p_{unstab}" ≥ 0.3, or "p_{illpo}" ≥ 0.1
- Wide input data range: "r_{coef}" $\geq 10^{10}$, "r_{rhs}" $\geq 10^{10}$, or "r_{obj}" $\geq 10^{10}$
- Inconsistent results: ("db" $-$ "pb")$/\max\{|$"db"$|, |$"pb"$|, 1\} > 10^{-6}$.

Table 7 Descriptions and references for numerically difficult test set

Example	Originator and description
alu10_1, alu10_5, alu10_7, alu10_8, alu10_9, alu16_1, alu16_2, alu16_5, alu10_7, alu16_8, alu16_9	T. Achterberg [1,2] Arithmetic logical unit (ALU) property checking instances. Feasible solutions correspond to counter-examples of given properties, infeasibility verifies correctness of property. The first number in each instance name is the number of input bits in the ALU. The second number indicates the property being checked. Properties 1–8 are valid, and property 9 is invalid
bernd2	T. Koch Wideband Code Division Multiple Access (W-CDMA) base station assignment problem
cnr_dual_mip1, cnr_heur_mip1, ilp_sh5, ilp_sh6, prodplan1, prodplan2, opti_157_0, p4, x01	Zuse Institute Berlin (ZIB) Instances from research projects at ZIB
dfn6_load, dfn6fp_load, dfn6f_cost, dfn6fp_cost	T. Koch [11] Access planning for German National Research and Education Network
neumshcherb	A. Neumaier, O. Shcherbina [35] Small numerically difficult instance given as example in [35]
norm-aim	Pseudo-Boolean Competition 2010 [38] Short for `normalized-aim-200-1_6-yes1-3` Instance from pseudo-Boolean competition at the SAT 2010 conference
npmv07, ns2017839, ran14x18.disj-8, sp98ir	MIPLIB 2010 [29] Instances from the MIPLIB 2010 library
neos-1053591, neos-1062641, neos-1367061, neos-1603965, neos-522351, neos-619167, neos-799716, neos-839838	COR@L [31] Instances from the COR@L test library
ns1629327, ns1770598, ns1859355, ns1866531, ns1900685, ns1925218, ns2080781	H. Mittelmann [18,34] Instances submitted to QSOPT_EX [7] through the NEOS server
tkat3K, tkat3T, tkat3TV, tkatTV5	T. Koch [28] Facility location problems from Telekom Austria

6.2 Computational study

We will discuss three topics on the numerically difficult instances: the error-proneness of the inexact solver, the performance of the exact solver, and the relevance of branching decisions based on exact LP solutions. The last point will be addressed in the next section on possible improvements.

Table 8 Selection criteria for numerically difficult test set

Example	p_{susp}	p_{unstab}	p_{illpo}	AL	r_{coef}	r_{rhs}	r_{obj}	db	pb
alu10_1	0.373	0.006	0.003	0.008	1e+06	1e+06	1	**Infeas**	**8.3999994e+01**
alu10_5	**0.536**	**0.582**	0.001	**0.179**	1e+06	1e+06	1	Infeas	Infeas
alu10_7	0.444	0.216	0.021	0.069	1e+06	1e+06	1	**9.1000004e+01**	**8.3000006e+01**
alu10_8	0.085	0.064	0.003	0.020	2.1e+06	2.1e+06	1	**8.7000000e+01**	**8.2999996e+01**
alu10_9	0.032	0.032	0.023	0.033	2.1e+06	2.1e+06	1	**9.3000000e+01**	**8.2999996e+01**
alu16_1	0.483	0.206	0.022	0.065	4.3e+09	4.3e+09	1	**Infeas**	**8.0000000e+01**
alu16_2	**0.614**	**0.971**	0.017	**0.294**	4.3e+09	4.3e+09	1	Infeas	Infeas
alu16_5	0.040	**1.000**	0.033	**0.300**	4.3e+09	4.3e+09	1	Infeas	Infeas
alu16_7	**0.932**	0.176	0.048	0.071	4.3e+09	4.3e+09	1	**9.9000000e+01**	**7.2000030e+01**
alu16_8	0.074	0.007	0.019	0.019	8.6e+09	8.6e+09	1	**8.6000000e+01**	**7.2000030e+01**
alu16_9	0.095	0.024	0.031	0.032	8.6e+09	8.6e+09	1	**1.2100000e+02**	**7.2000008e+01**
bernd2	**0.889**	0.121	0.003	0.045	**1.8e+10**	5e+08	2e+04	**1.1309147e+05**	**1.0858925e+05**
cnr_dual_mip1					4.3e+06	2.1e+09	**2e+11**	5.9803578e+07	5.9803578e+07
cnr_heur_mip1					4.3e+06	2.1e+09	**2.4e+13**	5.9803579e+07	5.9803579e+07
dfn6_load	0.016			0.000	5.8e+06	1e+04	4	**4.3728774e+00**	**3.7438423e + 00**
dfn6fp_load	0.113	0.001		0.001	8.6e+06	1e+04	1.7e+06	**7.6974176e+00**	**6.9360751e+00**
dfn6f_cost	0.062	0.012		0.004	8.6e+06	1e+04	4	**1.0000000e+03**	**9.0000000e+02**
dfn6fp_cost	0.070	0.002		0.001	8.6e+06	1e+04	1.7e+08	**1.0000131e+03**	**9.0001138e+02**
ilp_sh5	0.021	0.050		0.015	**1.8e+10**	1.7e+06	4	1.4280000e+03	1.4280000e+03
ilp_sh6	0.090	0.047		0.014	**1.8e+10**	1.7e+06	4	1.4120000e+03	1.4120000e+03
neumshcherb	**0.500**			0.005	17	4.6	1	**Infeas**	**-2.0000000e+00**
norm-aim	0.282	**0.624**	0.064	**0.246**	2	1	1	**Infeas**	**1.7700000e+02**
npmv07	0.378	**1.000**		**0.300**	2.3e+08	7.7e+04	1	1.0480981e+11	1.0480981e+11
neos-1053591	**0.650**			0.007	**1e+10**	1e+05	1	−3.6629144e+03	−3.6629144e+03
neos-1062641	**0.542**			0.005	3.6e+07	8.9e+05	1	2.4431301e−10	−7.8671292e−11

Table 8 continued

Example	p_{susp}	p_{unstab}	p_{illpo}	AL	r_{coef}	r_{rhs}	r_{obj}	db	pb
neos-1367061	**0.940**			0.009	8e+03	1	3.3e+02	3.1320456e+07	3.1320456e+07
neos-1603965	0.021			0.000	**2e+11**	1.5e+06	1.1e+04	**6.1947841e+08**	**6.1924437e+08**
neos-522351	**0.917**			0.009	5e+05	4e+03	1.6e+02	1.7891077e+04	1.7891077e+04
neos-619167	**0.907**	0.018	0.005	0.019	1.9e+07	1e+06	1	**2.1415929e+00**	**1.6648936e+00**
neos-799716	**1.000**	**0.939**		**0.282**	**1.7e+11**	1e+08	1	**Infeas**	**4.9326707e+06**
neos-839838	0.022			0.000	1e+08	1	3.1e+06	**1.0667738e+08**	**1.0665717e+08**
ns1629327					13	8.9	**2.7e+11**	−1.0980319e+01	−1.0980319e+01
ns1770598	0.105	0.005	0.001	0.003	**8.1e+13**	**8.2e+15**	1	2.5968209e+04	2.5968209e+04
ns1859355	0.048			0.000	1.5e+07	1.3e+04	1	**8.0467056e+00**	**7.9945700e+00**
ns1866531	**0.500**			0.005	8.2e+07	9e+04	1	**9.0000001e+00**	**8.9335230e−07**
ns1900685	**0.912**			0.009	3.2e+06	**1.2e+12**	1.4e+02	**3.4530000e+03**	**1.1590000e+03**
ns1925218	0.083	**0.883**	**0.117**	**0.382**	1e+09	1e+09	1	**Infeas**	**4.6156758e+06**
ns2080781	0.118	0.059		0.019	**2.7e+11**	1e+06	1	1.6029844e−13	**0.0000000e+00**
ns2017839	**1.000**	0.094		0.037	1.1e+08	1e+07	8.3e+06	**Infeas**	**7.7030495e+13**
opti_157_0			**0.167**	**0.167**	**1e+21**	**2e+21**	1	**Infeas**	**8.5931330e+03**
p4	0.301			0.003	1.4e+06	3.5e+06	**2.9e+12**	2.0226653e+14	2.0226653e+14
x01	**1.000**			0.010	2.1e+06	2.2e+06	1.2e+08	2.1178284e+10	2.1476524e+10
prodplan1	**1.000**	**1.000**		**0.300**	4.6e+06	**5.6e+12**	1.1e+05	−5.4578562e+07	−5.4578562e+07
prodplan2	**1.000**			0.010	4.2e+04	**5.9e+10**	1.7	−2.3939943e+05	−2.3939943e+05
ran14x18.disj-8				0.000	**1.2e+10**	45	2.6e+02	**3.7610000e+03**	**3.7120000e+03**
sp98ir					5.7e+02	3.6e+03	93	**2.1996003e+08**	**2.1967679e+08**
tkat3K	**0.904**			0.009	8e+04	1.5e+05	1.2e+05	4.7728181e+06	4.7728181e+06
tkat3T	**0.949**	0.001		0.010	8e+04	1.5e+05	2.3e+05	5.5648918e+06	5.5648918e+06
tkat3TV	**0.920**	0.001		0.009	8e+04	1.5e+05	2.3e+05	8.3883987e+06	8.3883987e+06
tkatTV5	**0.975**			0.010	1.6e+05	1.2e+05	7.8e+05	2.8117644e+07	2.8117644e+07

Bold numbers indicate that criterion is met

For this purpose, we evaluated the results and performance of our best exact branch-and-bound version of SCIP ("Exact-Reliability" with reliability branching and dual bounding strategy: automatic selection interleaved by exact LP calls) and its inexact counterpart ("Inexact-Reliablity"). The set-up of the experiment is the same as for the easy test set, described in Sect. 3.6; in particular, we use a time limit of 24 h and a memory limit of 13 GB. Note that during the test set selection, very hard instances for which both solvers failed to terminate within the imposed limits were removed.

First, we check how often the inexact run produced wrong results. Like Table 6 for the easy instances, Table 9 presents the absolute "Difference" between the objective function values returned by the exact and the inexact run ("Exact Objval" and "Approx Objval"). Since `bernd2` and `ns1770598` have very long exact objective function values, we only print the number of digits of their numerators and denominators. Again, we mark cases where a solver did not terminate at optimality within the limits (see Table 11 discussed later) by a "—". For non-zero values, we report in column "Difference" the closest FP-number.

We can compare the results for 26 instances, on the others, one of the solvers did not terminate within the limits. For half of them, the returned objective values were different, where for five instances (`alu10_1`, `bernd2`, `dfn6fp_cost`, `ns1866531`, and `opti_157_0`) this difference was significant. Furthermore, it is known that except for `alu10_9` and `alu16_9` all of the `alu` instances in our test set are infeasible, meaning the inexact run fails on at least five more instances.

Now we evaluate the performance. On the easy test set, the exact solver was only moderately slower than the inexact one and could solve all but one instance within the limits. In geometric mean, the solution time doubled, where most instances were only up to 20 times slower and the largest slowdown factor was 40. The node count in geometric mean was similar in the exact and the inexact branch-and-bound runs. Here, the picture is more diverse. Table 11 presents the solution times and the node count for the individual instances; they are split into four subsets depending on the accuracies of the inexact solver (zero, small, significant ($> 10^{-6}$), or unknown "Difference" in Table 9). The results are summarized in Table 10 and visualized in Fig. 4. They have the same layout as the tables and plots in Sect. 3.6.

First of all, on 9 instances, we actually benefit from taking care of the numerics. There were 4 instances (`alu16_5`, `cnr_dual_mip1`, `p4`, and `x01`) that were solved within the limits by the exact solver, but not by the inexact one, and 5 instances (`norm-aim`, `neos-839838`, `sp98ir`, `tkat3T`, and `tkat3TV`) that were solved by both versions, but where the exact solver was faster; the speed factors vary between 1.5 and 15.

We now analyze the other 41 instances, where the exact code is slower than the inexact one. Notice that, by definition of the numerically difficult test set (it contains only instances which one of the solvers can process within the imposed limits), the inexact code terminates on all these instances. We first observe that the exact code can solve only 21 instances, a much smaller portion than on the easy test set. Here, the degradation factors for the time are in most cases only up to 20 as well; but we also observe larger factors of up to 300 (`alu10_1`, `neos-1053591`, and `ns1629327`). However, for `alu10_1` the inexact solver returned a wrong answer. Examining the

Table 9 Comparison of exact and approximate objective function values on *numerically difficult test set*

Example	Exact-Reliablity Exact Objval	Inexact-Reliablity Approx Objval	Difference
alu10_1	Infeas	8.50000000000000e+01	∞
alu10_5	Infeas	Infeas	
alu10_7	—	8.30000047702360e+01	—
alu10_8	—	8.40000019080471e+01	—
alu10_9	—	8.40000019077561e+01	—
alu16_1	—	9.10000000000000e+01	—
alu16_2	Infeas	Infeas	
alu16_5	Infeas	—	—
alu16_7	—	7.90000000000000e+01	—
alu16_8	—	7.90000000008731e+01	—
alu16_9	—	7.90000000000000e+01	—
bernd2	(int of 2147 digits)/(int of 2141 digits)	1.12090603170608e+05	1.00086584527058e+03
cnr_dual_mip1	1196071565864636 27/2000000000	—	—
cnr_heur_mip1	—	5.98035789799395e+07	—
dfn6_load	—	3.74384225843200e+00	—
dfn6fp_load	—	6.93593657927233e+00	—
dfn6f_cost	1000	1.00000000000000e+03	
dfn6fp_cost	5000065343183/5000000000	1.00001080799110e+03	2.26064550000000e-03
ilp_sh5	1428	1.42800000000000e+03	
ilp_sh6	1412	1.41200000000277e+03	2.77000000000000e-09
neumshcherb	-2	-2.00000000186265e+00	1.86265000000000e-09
norm-aim	200	2.00000000000000e+02	
npmv07	—	1.04809812554514e+11	—
neos-1053591	-4578643/1250	-3.66291440000000e+03	
neos-1062641	—	0.00000000000000e+00	—
neos-1367061	783011406612429/25000000	3.13204562644972e+07	4.00000000000000e-08
neos-1603965	—	6.19244367662955e+08	—
neos-522351	4472769279/250000	1.78910771160000e+04	
neos-619167	—	1.67926829831853e+00	—
neos-799716	—	4.93267066169203e+06	—
neos-839838	2133143427299/20000	1.06657171364950e+08	
ns1629327	-109803191329325384099/1000000000000000000	-1.09803191329325e+01	3.84099000000000e-14
ns1770598	(int of 1190 digits)/(int of 1184 digits)	2.59682092873858e+04	3.20057499299592e-08
ns1859355	17656324 3137728817544965354640796968772247963283039222284232036674193433465271403875/ 219423017121743547661843936752008732791214806785323560158027452138269863320820 11659	8.04670564893240e+00	9.28049536228506e-12
ns1866531	10	9.50234212056133e-07	9.99999904976579e+00
ns1900685	3453	3.45300000000000e+03	
ns1925218	—	6.86828679741287e+06	—
ns2080781	—	0.00000000000000e+00	—
ns2017839	—	7.70304949632221e+13	—
opti_157_0	Infeas	8.59313300000015e+03	∞
p4	Infeas	—	—
x01	Infeas	—	—
prodplan1	—	-5.45785617339179e+07	—
prodplan2	-185497557187228867655290921686042318951/ 774845425502612687685652190000000	-2.39399435141407e+05	2.99563575827916e-10
ran14x18.disj-8	—	3.71199999853663e+03	—
sp98ir	1098383952/5	2.19676790400000e+08	
tkat3K	47728181/10	4.77281810000000e+06	
tkat3T	22259567/4	5.56489175000000e+06	
tkat3TV	167767973/20	8.38839864999998e+06	2.00000000000000e-08
tkatTV5	1124705769/40	2.81176442250000e+07	

If absolute difference (computed exactly) between "Exact Objval" and "Approx Objval" is non-zero, closest FP number is displayed in "Difference"

remaining 20 instances, which were not solved by the exact code within the imposed limits, we already see that they will include even larger slowdown factors. But some of the results of the inexact solver will be wrong (for five `alu` instances this is already

Table 10 Summary of performance for best exact solver and inexact counterpart on *numerically difficult test set*

Setting	slv	Geometric mean for instances solved by all settings (26)		
		Nodes	Time (s)	DB (s)
Inexact-Reliability	46	5,650	93.7	–
Exact-Reliability	30	10,499	368.4	58.8

"slv" number of instances solved, "DB" safe dual bounding time

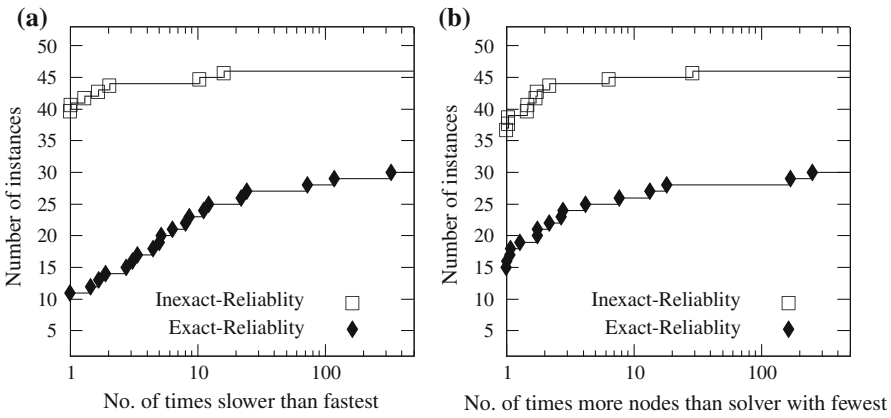

Fig. 4 Comparison of best exact solver and inexact counterpart on *numerically difficult test set*. **a** Performance profile for overall solving time "Time". **b** Performance profile for branch-and-bound nodes "Nodes"

known for sure, see above), since most of these instances were collected because of inconsistent results between different solvers and settings.

Why is the performance not as good as on the easy test set? For the numerically more difficult instances, the exact code has to often process more branch-and-bound nodes. As explained in Sect. 4, this is to some extend due to reliability branching being sensitive to small changes in the solving process, but the main reason is that the inexact solver *wrongly* cuts off some nodes due to FP-errors. Table 11 presents, in Columns "NotInfeas", "NotInt", and "NotInt-Inf", how often the exact code would have made wrong decisions if the result of the inexact LP solver would not have been safely verified; which indicates wrong decisions in the inexact MIP solver. All larger slowdown factors come along with mistakes in the inexact solver; except for `dfn6fp_load`, `ns2017839` and `prodplan1`, where the degradation is caused by expensive LP calls.

Column "NotInfeas" states the number of nodes where the inexact LP solver *wrongly* claims LP infeasibility at a node, which leads to more branchings in the exact solver and thus increases the node count. This happens on 9 of the 50 instances, but never occurred on the easy MIPs.

Column "NotInt" counts the nodes where the floating-point LP solution was integral within tolerances (i.e., would have been accepted by the inexact solver) but verifying the primal bound (Step 7 of Algorithm 1) did not allow us to cut off the node. This

Table 11 Overall performance of best exact solver and inexact counterpart on *numerically difficult test set*

Example	Inexact-Reliability		Exact-Reliability				
	Nodes	Time (s)	Nodes	Time (s)	NotInt	NotInt-Inf	NotInfeas
alu10_5	4,077	**3.9**	74,183	95.7			
alu16_2	59	**1.0**	63	5.1			4
×dfn6f_cost	23,771	**6452.8**	320,840	56063.3	384	49	
×ilp_sh5	124,213	**3433.8**	126,235	21547.7			
norm-aim	169,635	164.5	5,863	**15.9**			2
×neos-1053591	88,969	**117.4**	241,386	8211.0	2,572	1,230	
×neos-522351	20,483	**35.7**	26,369	420.6			
×neos-839838	71,119	7235.0	40,867	**4365.5**			
×ns1900685	29,336	**4.3**	28,438	8.2			
sp98ir	78,788	6615.5	12,385	**410.9**			
tkat3K	3,469	**16.1**	6,131	44.5			
tkat3T	13,490	108.4	9,220	**83.5**			
tkatTV5	10,113,175	**29585.9**	5,947,286	**29845.5**			
×ilp_sh6	16,083	**594.6**	15,513	1864.4			
neumshcherb	5	**1.0**	5	**1.0**			
×neos-1367061	267	**1065.7**	745	5356.3			
×ns1629327	26,138	**26.5**	57,051	2772.2	273		
ns1770598	11,519	**36.9**	7,956	53.9			
×ns1859355	30,396	**65.3**	32,936	221.3	298	87	
×prodplan2	4	**1.0**	31	26.0			
tkat3TV	16,796	147.9	7,689	**72.7**			
alu10_1	5,859	**5.9**	1,489,343	1930.5		*5,189*	
×bernd2	13,405	**10049.0**	23,488	81808.3	178	75	
×dfn6fp_cost	16,921	**2022.6**	71,039	9056.4	53	48	
ns1866531	1	**1.0**	170	13.7	91	64	
×opti_157_0	119	**4.1**	119	8.2		*1*	
alu10_7	2,959	**2.5**	>87,559,918	>86400.0	*≥1,227*	*≥520*	
alu10_8	40,372	**21.1**	>75,755,192	>86400.0	*≥941*	*≥441*	
alu10_9	94,144	**45.3**	>74,933,946	>86400.0	*≥444*	*≥428*	
alu16_1	2,783	**4.4**	>41,441,026	>86400.0	*≥42,807*	*≥496,233*	≥246,605
alu16_5	>171,786,359	>86400.0	101,727	**161.9**			1,059
alu16_7	3,469	**3.6**	>4,330,018	>86400.0	*≥294,094*	*≥258,965*	≥2,348
alu16_8	1,415,760	**980.4**	>98,010,563	>86400.0	*≥7,310*	*≥717*	≥2,687
alu16_9	829,156	**432.6**	>16,800,387	>86400.0	*≥410,640*	*≥36,361*	≥490
×cnr_dual_mip1	>1,612,128	>86400.0	82,332	**11523.5**			
×cnr_heur_mip1	321,303	**38523.2**	>935,604	>86400.0			
×dfn6_load	2,846	**37.1**	>136,256	>86400.0	≥66,286	≥11,624	
×dfn6fp_load	44,538	**8211.8**	>1,671	>86400.0			

Table 11 continued

Example	Inexact-Reliability		Exact-Reliability				
	Nodes	Time (s)	Nodes	Time (s)	NotInt	NotInt-Inf	NotInfeas
×npmv07	46	**26.2**	>53	>86400.0	*≥19*		
×neos-1062641	100	**1.0**	>598,633	>86400.0		*≥299,253*	
×neos-1603965	1	**61.7**	>1,872	>86400.0	*≥1,872*		
×neos-619167	246,133	**7909.1**	>784,177	>86400.0		*≥38,242*	
×neos-799716	115	**74.6**	>6,013	>86400.0	≥25		≥3
×ns1925218	886,119	**13804.4**	>3,699,549	>86400.0			≥70,065
ns2080781	40	**1.0**	>2,373,236	>86400.0	*≥211,271*	*≥598,285*	
×ns2017839	32	**543.8**	>1	>86400.0			
×p4	>1,529,624	>86400.0	1	**191.8**			
×x01	>729,677	>86400.0	1	**104.4**			
×prodplan1	100,628	**26804.9**	>13	>86400.0			
ran14x18.disj-8	22,477,620	**52701.6**	>31,821,206	>86400.0	≥16,801	≥9	

Detailed results, grouped by accuracy of the inexact solver (zero, small, significant, or unknown "Difference" in Table 9). "NotInt" plus "NotInt-Inf" ("NotInfeas") counts nodes where LP relaxation was wrongly claimed integral (infeasible) by floating-point LP solver; italic font if all integrality claims were wrong. Instances missing bounds on variables are marked by "×" . Solving times within 5% of the fastest setting are put in bold

happens on 20 of the 50 instances, in contrast to only one instance for the easy test set (`markshare1_1`, where "NotInt" is 256). Note that "NotInt" only considers nodes where branching on the exact LP solution takes place afterwards. That is, approximate integral LP solutions for which the corresponding exact LP turns out to be infeasible (so pruning is legal but the argumentation of the inexact solver is wrong) are not counted here but in Column "NotInt-Inf". A rejected approximate primal solution does not only mean that we can not cut off the current subtree in the exact code, but it may also affect other parts of the tree because the primal bound in the exact code is weaker than the, possibly incorrect, bound in an inexact solver. In the extreme case, this leads to rejecting all approximate solutions found and we are not able to cut off any node early by bounding; an italic font in Columns "NotInt" and "NotInt-Inf" indicates such cases. The unsolved `alu` instances, `npmv07`, `neos-1062641`, and `ns2080781`, all with extreme degradation factors, are examples for this effect.

6.3 How to tackle numerically difficult instances?

All in all, the exact code is slower on the numerically difficult test set, sometimes requiring much more time to solve an instance, or even failing to finish within the imposed limits. However, a direct comparison of the solution times is not always fair here because the inexact solver frequently, in particular, on instances with huge differences in the performance, takes advantage of incorrect bounding decisions.

Introducing presolving, cutting planes, and primal heuristics will certainly help to improve the performance as it normally shrinks the size of the branch-and-bound tree

and thus reduces the space of the search tree which the inexact solver would incorrectly ignore, but the exact code has to process.

In addition to the generally increased node count, the time overhead also comes from the exact LP solves in the safe primal bounding step and the ones for disproving LP infeasibility of nodes. On the numerically difficult instances, such exact LP solves are more often experienced or they occur so often that they add up to a large portion of the running time. Thus, more sophisticated methods for the safe primal feasibility check are required.

The current solver uses the first fractional variable branching rule when it branches on the exact LP solution. This type of branching happens in two situations. First, if the approximate LP solution is nearly integral, but the safe primal bounding step (where the exact LP is warm started with the basis of the approximate LP solution) does not allow to prune the node (the computed exact LP solution is not integral). Second, if the LP relaxation is claimed to be infeasible, but there exists an exact LP solution. Our fast safe dual bounding methods are useless here, we have to solve this LP exactly to prove LP feasibility. In contrast to the easy test set, both situations occur frequently on the numerically difficult test set; numbers were given in Table 11 in Columns "NotInt" and "NotInfeas". Furthermore, both situations can easily occur again in the subtrees created after branching on the exact LP solution. A branching rule that reduces the risk of such expensive situations for the new subtrees could be helpful for numerically difficult instances.

7 Conclusion

From the computational results we can make several key observations. Each safe 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 57 problems from the easy test set solved within 2 h by the floating-point branch-and-bound solver, 55 could also be solved exactly within 24 h and the solution time was usually no more than 20 times slower. This demonstrates that the hybrid methodology can lead to an efficient exact branch-and-bound solver, not limited to specific classes of problems.

When turning to numerically more difficult instances, where floating-point solvers face numerical troubles and even compute incorrect results, we observe some stronger slowdowns with our current exact solver. However, this mainly happens on instances where the inexact MIP solver strongly benefits from incorrect bounding decisions. As a consequence, the bottleneck of the exact solver is a large number of nodes for which the hybrid rational/safe floating-point approach cannot skip the expensive rational computations of the main procedure by replacing them with certain decisions from the faster slave procedure with FP-arithmetic. Examples are wrong infeasibility detections of the floating-point LP solver and incorrect integrality claims based on the approximate LP result. In the future, we will investigate techniques to process such nodes without calling exact LP solvers and how to prevent situations like this from repeating in subsequent subtrees.

Acknowledgments The authors would like to thank Tobias Achterberg for helpful discussions on how to best incorporate the exact MIP features into SCIP. We would also like to thank Daniel Espinoza for his assistance with QSOPT_EX, which included adding new functionalities and writing an interface for use within SCIP.

References

1. Achterberg, T.: ALU instances. http://miplib.zib.de/miplib2003/contrib/ALU
2. Achterberg, T.: Constraint Integer Programming. Ph.D. thesis, Technische Universität Berlin (2007)
3. Achterberg, T.: SCIP: Solving constraint integer programs. Math. Program. Comput. **1**(1), 1–41 (2009)
4. Achterberg, T., Koch, T., Martin, A.: The mixed integer programming library: MIPLIB (2003). http://miplib.zib.de
5. Althaus, E., Dumitriu, D.: Fast and accurate bounds on linear programs. In: Vahrenhold, J. (ed.) SEA 2009. LNCS, vol. 5526, pp. 40–50. Springer, Berlin (2009)
6. Applegate, D.L., Bixby, R.E., Chvátal, V., Cook, W.J.: The Traveling Salesman Problem: A Computational Study. Princeton University Press, Princeton (2006)
7. Applegate, D.L., Cook, W.J., Dash, S., Espinoza, D.G.: QSopt_ex. http://www.dii.uchile.cl/daespino/ESolver_doc/main.html
8. Applegate, D.L., Cook, W.J., Dash, S., Espinoza, D.G.: Exact solutions to linear programming problems. Oper. Res. Lett. **35**(6), 693–699 (2007)
9. AT&T Bell Laboratories, The University of Tennessee Knoxville, and Oak Ridge National Laboratory. Netlib Repository. http://www.netlib.org/netlib/lp
10. Bixby, R.E., Ceria, S., McZeal, C.M., Savelsbergh, M.W.: An updated mixed integer programming library: MIPLIB 3.0. Optima **58**, 12–15 (1998)
11. Bley, A., Koch, T.: Optimierung des G-WiN. DFN-Mitteilungen **54**, 13–15 (2000)
12. Chai, J., Toh, K.: Computation of condition numbers for linear programming problems using Peña's method. Optim. Methods Softw. **21**(3), 419–443 (2006)
13. Cheung, D., Cucker, F.: A new condition number for linear programming. Math. Program. **91**, 163–174 (2001)
14. Cheung, D., Cucker, F.: Solving linear programs with finite precision: I. Condition numbers and random programs. Math. Program. **99**, 175–196 (2004)
15. Cheung, D., Cucker, F.: Solving linear programs with finite precision: II. Algorithms. J. Complex. **22**(3), 305–335 (2006)
16. Cheung, D., Cucker, F., Peña, J.: Unifying condition numbers for linear programming. Math. Oper. Res. **28**(4), 609–624 (2003)
17. Cook, W.J., Dash, S., Fukasawa, R., Goycoolea, M.: Numerically safe Gomory mixed-integer cuts. INFORMS J. Comput. **21**(4), 641–649 (2009)
18. Czyzyk, J., Mesnier, M.P., Moré, J.J.: The Network-Enabled Optimization System (NEOS) server. IEEE J. Comput. Sci. Eng. **5**(3), 68–75 (1998)
19. de Vries, S., Vohra, R.: Combinatorial auctions: a survey. INFORMS J. Comput. **15**(3), 284–309 (2003)
20. Dhiflaoui, M., Funke, S., Kwappik, C., Mehlhorn, K., Seel, M., Schömer, E., Schulte, R., Weber, D.: Certifying and repairing solutions to large LPs, how good are LP-solvers? In: SODA 2003, pp. 255–256. ACM/SIAM, New York (2003)
21. Dolan, E.D., Moré, J.J.: Benchmarking optimization software with performance profiles. Math. Program. **91**(2), 201–213 (2001)
22. Espinoza, D.G.: On Linear Programming, Integer Programming and Cutting Planes. Ph.D. thesis, Georgia Institute of Technology (2006)
23. Gay, D.M.: Electronic mail distribution of linear programming test problems. Math. Program. Soc. COAL Newsl. **13**, 10–12 (1985)
24. GMP. GNU multiple precision arithmetic library. http://gmplib.org
25. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. (CSUR) **23**(1), 5–48 (1991)
26. IBM, ILOG. CPLEX. http://www.ilog.com/products/cplex
27. Koch, T.: The final NETLIB-LP results. Oper. Res. Lett. **32**(2), 138–142 (2004)
28. Koch, T.: Rapid Mathematical Programming. Ph.D. thesis, Technische Universität Berlin (2004)

29. Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R.E., Danna, E., Gamrath, G., Gleixner, A.M., Heinz, S., Lodi, A., Mittelmann, H., Ralphs, T., Salvagnin, D., Steffy, D.E., Wolter, K.: MIPLIB 2010. Math. Program. Comput. **3**(2), 103–163 (2011)
30. Kwappik, C.: Exact Linear Programming. Universität des Saarlandes, Master thesis (1998)
31. Lehigh University. COR@L mixed integer programming collection. http://coral.ie.lehigh.edu/data-sets/mixed-integer-instances
32. Linderoth, J.T., Ralphs, T.K.: Noncommercial software for mixed-integer linear programming. In: Karlof, J. (ed.) Integer programming: theory and practice, pp. 253–303. CRC Press, Boca Raton (2005)
33. Mittelmann, H.D.: Benchmarks for Optimization Software (2010). http://plato.asu.edu/bench.html
34. NEOS Server. http://neos-server.org/neos
35. Neumaier, A., Shcherbina, O.: Safe bounds in linear and mixed-integer linear programming. Math. Program. **99**(2), 283–296 (2004)
36. Nunez, M., Freund, R.M.: Condition measures and properties of the central trajectory of a linear program. Math. Program. **83**, 1–28 (1998)
37. Ordónez, F., Freund, R.M.: Computational experience and the explanatory value of condition measures for linear optimization. SIAM J. Optim. **14**(2), 307–333 (2003)
38. Pseudo-Boolean Competition (2010). http://www.cril.univ-artois.fr/PB10/
39. Renegar, J.: Some perturbation theory for linear programming. Math. Program. **65**, 73–91 (1994)
40. Renegar, J.: Incorporating condition measures into the complexity theory of linear programming. SIAM J. Optim. **5**, 506–524 (1995)
41. Renegar, J.: Linear programming, complexity theory and elementary functional analysis. Math. Program. **70**, 279–351 (1995)
42. Steffy, D.E.: Topics in Exact Precision Mathematical Programming. Ph.D. thesis, Georgia Institute of Technology (2011)
43. Steffy, D.E., Wolter, K.: Valid linear programming bounds for exact mixed-integer programming. Technical Report ZR 11–08, Zuse Institute Berlin. INFORMS J. Comput. (2011, to appear)
44. Vera, J.R.: On the complexity of linear programming under finite precision arithmetic. Math. Program. **80**, 91–123 (1998)
45. Wilken, K., Liu, J., Heffernan, M.: Optimal instruction scheduling using integer programming. In: ACM SIGPLAN 2000, vol. 35, pp. 121–133. ACM Press, New York (2000)
46. Zuse Institute Berlin. SCIP. http://scip.zib.de
47. Zuse Institute Berlin. SoPlex. http://soplex.zib.de