Analyzing the numerical correctness of branch-and-bound decisions for mixed-integer programming

Alexander Hoen[1] and Ambros Gleixner[1,2]

[1] Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
hoen@zib.de
[2] HTW Berlin, 10313 Berlin, Germany
gleixner@htw-berlin.de

Abstract. Most state-of-the-art branch-and-bound solvers for mixed-integer linear programming rely on limited-precision floating-point arithmetic and use numerical tolerances when reasoning about feasibility and optimality during their search. While the practical success of floating-point MIP solvers bears witness to their overall numerical robustness, it is well-known that numerically challenging input can lead them to produce incorrect results. Even when their final answer is correct, one critical question remains: Were the individual decisions taken during branch-and-bound justified, i.e., can they be verified in exact arithmetic? In this paper, we attempt a first such *a posteriori* analysis of a pure LP-based branch-and-bound solver by checking all intermediate decisions critical to the correctness of the result: accepting solutions as integer feasible, declaring the LP relaxation infeasible, and pruning subtrees as suboptimal. Our computational study in the academic MIP solver SCIP confirms the expectation that in the overwhelming majority of cases, all decisions are correct. When errors do occur on numerically challenging instances, they typically affect only a small, typically single-digit, amount of leaf nodes that would require further processing.

Keywords: Mixed integer programming, branch and bound, exact computation

1 Introduction

Mixed Integer Programming (MIP) solvers have evolved into highly sophisticated software capable of solving a growing number of large-scale and complex problems efficiently. They typically implement a branch-and-bound scheme based on a linear programming (LP) relaxation. For performance reasons, nearly all solvers utilize floating-point arithmetic in order to speed up computations and control memory requirements for representing values of rational numbers. These computational advantages of floating-point arithmetic, however, also imply limitations in precision: not all rational numbers can be represented exactly, leading to tiny round-off errors that can accumulate in magnitude over the course of the algorithm. Therefore, solvers introduce tolerances for feasibility and a zero tolerance for deciding equality between two numbers. Additionally, by default, commercial solvers operate with relative gaps between 0.01% and 0.0001%.

It is well documented that sometimes accumulated rounding errors can cause solvers to incorrectly declare infeasibility, feasibility, or optimality for suboptimal solutions [6,15,19,22]. Further, the MIPLIB 2017 instance collection labels 193 instances with problematic numerics with a tag “numerics”, partly because different solvers reported inconsistent results during tests for the compilation process [12]. As a result, for applications that require absolute certainty, such as chip design verification [1], combinatorial auctions [24], or computational proofs in experimental mathematics [10], the level of trust provided by floating-point solvers is often not sufficient.

Nevertheless, it is widely accepted that the way that mature floating-point MIP solvers handle numerics is overall very robust. This judgment is not only due to the fact that for the vast majority of industrial MIP applications, it may be acceptable to work with solutions that are slightly suboptimal or slightly infeasible. The wide-spread use of floating-point MIP solvers and the fact that the general methodology of handling round-off errors in MIP solvers has not changed substantially over many years of development rather suggests that in the vast majority of cases, their answers may indeed be correct in an exact sense.

However, to the best of our knowledge, the literature holds no record of a systematic computational investigation of this fundamental methodological question: To what extent do approximate floating-point solvers provide numerically exact answers? More specifically, to what extent is the *reasoning* of a floating-point LP-based branch-and-bound solver, which is composed of a large number of critical decisions during the search, correct in an exact sense? While studies exist that compare the final result of numeric and exact solvers, e.g., in [7], we are not aware of any analysis that looks deeper at the branch-and-bound process. This is particularly interesting for the large number of cases where the final answer of a floating-point solver is correct: Is this due to the fact that all individual decisions were correct, or because all incorrect decisions did not affect or compromise the final result?

In this paper, we try to give empirical answers to these questions by performing a first such *a posteriori* analysis of a pure LP-based branch-and-bound process. To this end, we inspect all leaves of the branch-and-bound tree and check if the leaf is pruned correctly because (a) its LP relaxation is infeasible, (b) its dual bound proves that the leaf does not contain improving solutions, or (c) the LP solution is primal feasible. To test the correctness of each decision, we apply a hierarchy of techniques from the literature such as safe bounding by directed rounding [17], rational reconstruction by continued fraction approximations [23,25,18,20], exact LU factorization of simplex bases, and, if necessary, round-off-error-free LP solving [13,14]. We base our experiments on the open-source MIP solver SCIP [3] and use two curated test sets from the literature with and without numerical challenges [7]. Our code base is publicly available.[3]

[3] https://github.com/alexhoen/bnbanalyzer

The paper is organized as follows. In Section 2, we briefly introduce the general concept of a branch-and-bound solver, provide a classification of numerical errors, and survey the techniques to check the correctness of floating-point decisions. In Section 3, we present the results of our experiments and analyze the different errors encountered in the floating-point solver. In Section 4, we summarize our results and give an outlook on their implications for future research.

2 Numerical Errors in LP-based Branch and Bound

A *mixed integer program* (MIP) is given in the form

$$\begin{aligned} \min\ & c^T x \\ & Ax \geq b \\ & \ell \leq x \leq u \\ & x_i \in \mathbb{Z} \ \text{ for all } i \in \mathcal{I}, \end{aligned}$$

where we assume rational input data $A \in \mathbb{Q}^{m \times n}$, $c \in \mathbb{Q}^n$, $u, l \in (\mathbb{Q} \cup \{\pm\infty\})^n$, and $b \in \mathbb{Q}^m$. The index set $\mathcal{I} \subseteq \{1, \dots, n\}$ defines which decision variables are required to be integer.

MIPs are typically solved by variants of *LP-based branch and bound*, which dates back to [16] and still forms the backbone of today's most competitive MIP solvers.

In Section 2.1, we describe briefly the procedure of an LP-based branch-and-bound algorithm. For a more detailed description please refer to [1]. In Section 2.3, we give an overview of the techniques we use to obtain an exact evaluation of a node that was solved in floating-point arithmetic. In Section 2.2, we categorize how numerics can impact the critical branch-and-bound decisions in a floating-point solver and how incorrect decisions affect the overall result.

2.1 Basics of the LP-based Branch-and-Bound Algorithm

The first step in an LP-based branch-and-bound algorithm is to *relax* the by dropping all integrality constraints. The resulting LP relaxation can be solved by an LP solver. If the LP solution is *integer feasible*, i.e., all integer variables have an integer solution value, the problem is solved. If the LP relaxation is detected to be infeasible, then also the original MIP is infeasible. Otherwise, the algorithm *branches*: It chooses an integer variable x_i with a fractional value $\hat{x}_i$ in the solution of the relaxed problem to create two new sub-problems by taking the original problem and adding the constraint $x_i \geq \lceil \hat{x}_i \rceil$ and $x_i \leq \lfloor \hat{x}_i \rfloor$, respectively. This ensures that the optimal solution is still preserved in at least one of the two sub-problems and the current LP solution is no longer feasible in both sub-problems. This process of LP solving and branching is iterated recursively.

If during this process an integer feasible solution is found its objective value can be used to the *primal bound*, which is the objective value of the best known solution encountered so far. This primal bound can be used to *prune*, i.e., remove from further consideration all open sub-problems with a greater or equal *dual bound*, which is given by the objective value of the sub-problem's LP relaxation.

This recursive procedures produces a search tree with sub-problems as nodes. A node is called *leaf* if it is not further branched on because it is a) integer feasible, b) LP infeasible, or c) it was pruned. Figure 1 presents an example of a branch-and-bound tree for a minimization problem. The solution of the relaxed LP at the node is displayed slightly above and to the right of the node, while its objective value is positioned on the left side. The leaf N_4 is integer feasible and appears to be also optimal. The leaf N_3 is infeasible and the node N_1 is cut off since branching on N_1 does not improve the solution of N_4 anymore.

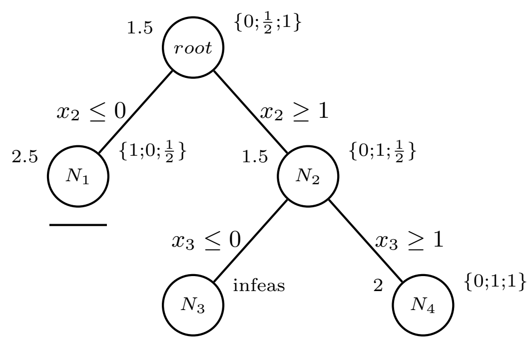

Fig. 1. An example for a branch-and-bound tree on a minimization problem.

Note that although the heuristic decision on which fractional variable to branch may be affected by numerical calculations, any sequence of branching decisions yields a correct branch-and-bound process. The same holds for the heuristic decision in which order to process open sub-problems: any order yields a correct branch-and-bound algorithm. By contrast, the decision of whether to remove a sub-problem from consideration is both affected by numerics and can critically affect the correctness of the final result. In the next section, we focus on these critical decisions.

2.2 Types of Numerical Errors in LP-based Branch and Bound

In our analysis, we distinguish the following types of incorrect decision that can be taken by a floating-point solver due to numerical errors. All of these errors occur at a leaf of the branch-and-bound tree.

Solution errors. By design, a floating-point solver may produce approximate solutions with slight violations of integrality and constraint feasibility. A simple post-processing step is to first round the value of all integer variables to the nearest integer and second, if continuous variables are present, compute rational values for these such that all constraints are satisfied exactly. If this is certifiably not possible, i.e., if the floating-point solver accepted an assignment for the integer variables as feasible that cannot be completed to an exactly feasible solution, we refer to this as a *solution error*.

We distinguish two types of solution errors. If the LP relaxation at this node is infeasible in exact arithmetic, then no solution was discarded by removing the current leaf. We refer to this error as a *weak solution error*. Otherwise, the primal solution of the LP relaxation does not satisfy integer feasibility, but the sub-problem below the current leaf node may still contain improving solutions. In this case, only a lower bound can be derived and we refer to this situation as a *strong solution error*.

Bound errors. If a floating-point solver uses a lower bound to prune a node, this decision can often be verified *a posteriori* by computing a safe dual bound that is valid in exact arithmetic, by techniques described in Section 2.3. If this safe bound is greater than or equal to the current primal bound, then the decision to prune the node was correct. Otherwise, if the exact objective value of the LP relaxation is below the primal bound, we label this decision as a *bound error*. Again, we distinguish two cases. If later during the solving process an exactly feasible solution is found that improves the primal bound and in hindsight justifies pruning the leaf, we call the error a *weak bound error*, otherwise a *strong bound error*.

Gap errors. For the last type of error, consider the situation that a floating-point solver declares a node to be integer feasible and produces an approximate solution that can be converted to an exactly feasible solution after rounding the integer assignment. Although we record no solution error here, the decision to terminate the search below this node may still be incorrect if the objective value of the converted, exact solution does not match the dual bound derived by solving the LP relaxation exactly. We refer to this error as a *gap error*, and as above we distinguish between *weak gap errors* that are justified in hindsight and *strong gap errors*.

Infeasibility errors. If the floating-point solver declares a leaf infeasible but there exists a solution to the LP relaxation that is exactly feasible we refer to this situation as an *infeasibility error*.

Even when some of the errors listed above occur, the overall result of the floating-point solver may still be correct. Specifically, weak bound and weak gap errors do not compromise the correctness of the final result since the corresponding decisions are justified in hindsight. We still record these errors, because we are interested not only in the correctness of the final result but in the correctness of the floating-point solver's reasoning that led to the result.

Also in the presence of strong bound or strong gap errors, the solution or at least its objective value z^* returned by the floating-point solver may still be optimal. Because our analysis produces a safe dual bound at all occurrences of such errors, we can then take the minimum of all these dual bounds and derive a global lower bound $\hat{z} < z^*$ for the true optimal objective value. This certifies that the true optimal objective value must lie in the interval $[\hat{z}, z^*]$. If infeasibility errors occur, i.e., if a leaf is removed although the LP relaxation is feasible in exact arithmetic, we can use the exact objective value of the LP relaxation as a valid dual bound.

Finally, note that in the presence of solution errors or gap errors, the floating-point solver will work with an incorrect, usually too optimistic primal bound during part of the search. When checking for bound errors, however, we always take the perspective of the floating-point solver, i.e., we treat the primal bound registered in the floating-point solver at that time as valid, and check whether the pruning decision is justified with respect to this primal bound. This helps us to separate more clearly and quantify more precisely the different types of incorrect reasoning in our computational analysis later.

2.3 Checking Numerical Correctness of Floating-point Decisions

A straightforward way to evaluate whether and which of the errors described above occurs at a leaf node is to solve the LP relaxation once again with an exact LP solver and compare the results. For most but small instances, this approach is prohibitively slow. Hence, we use a hierarchy of methods that may be able to verify or refute the correctness of leaf decisions faster.

If the plain floating-point verification fails we apply the following steps:

Safe bounding. In [17], Neumaier and Shcherbina introduce *safe bounding*: We assume we have an approximated dual solution $\{\hat{y}, \hat{r}^-, \hat{r}^+\}$ with the error

$$\varepsilon = c - A^T\hat{y} + I\hat{r}^+ + I\hat{r}^- \; . \tag{1}$$

We replace

$$r_i^- := \hat{r}_i^- + \min\{\varepsilon_i, 0\} \text{ and } r_i^+ = \hat{r}_i^+ + \max\{\varepsilon_i, 0\} \tag{2}$$

for all variables $i \in N$ and obtain a feasible dual solution $\{\hat{y}, r^-, r^+\}$ $b^T\hat{y} - l^T r^+ + u^T r^-$. This transformation ensures feasibility at the cost of reducing the objective value by $l^T(r^+ - \hat{r}^+) + u^T(\hat{r}^- - r^-)$ which can now serve as a valid lower bound for the primal objective. Since the quality of this lower bound is impacted by the variable bounds u, l these bounds have to be finite and, ideally, as tight as possible to minimize their impact. Safe-bounding can be applied analogously for validating the Farkas proof. This approach is also implemented for validating floating-point LP solutions and explained in more detail in [21].

Rational reconstruction We apply continued fractions approximation computed by the extended Euclidean algorithm to re-construct an approximated solution to an exact solution. Specifically, it uses the continued fraction expansion of the floating-point number to round it to a close rational with a limited denominator.This method is applied to both, the primal and the dual solution, as well as to the Farkas proof. We rely on the implementation of SoPlex [13,14]. This method has been introduced and improved by [23,25,18,20].

Factorization If available, we perform a rational LU factorization on the simplex basis achieved from the floating-point LP solver. For this, we use the implementation in SoPlex [13,14]. This step is only skipped for infeasible nodes since no basis can be obtained from the LP solver.

Exact LP solving The last and most time-expensive option to derive the most accurate lower bound on the objective is to call a round-off-error-free LP solver. The open-source solver SoPlex provides a configuration to perform such round-off-error-free LP solver [13,14]. We refer to this configuration as Exact-SoPlex. This is performed as the last option for all cases.

For obtaining a primal solution from an inaccurate floating-point solution, we also can fix the integer variables to the values of the floating-point solution and solve the LP over the continuous variables in rational arithmetic with Exact-SoPlex. For each primal solution value, it is ensured to lie within its lower and upper bounds.

3 Computational Study

The guiding questions for the design of our experiments were the following: Assuming the final result produced by a floating-point branch-and-bound solver is correct, can we also verify that the reasoning of the individual decisions of the solver is correct? If not, how many of the critical decisions would require re-evaluation, and which types of the errors defined in Section 2.2 occur at which frequency? Furthermore, we are interested in the effectiveness of the different post-processing techniques outlined in Section 2.3. Before evaluating our experiments in Section 3.2, we first describe the setup and specify the software used in Section 3.1.

3.1 Experimental Setup

As branch-and-bound framework, we use the open-source MIP solver SCIP 9.3 [1,3] (hash: b7906251d5) and configure SCIP via parameters such that it mimics the behavior of the pure LP-based branch-and-bound algorithm described in [6]. We use the open-source solver SoPlex 8.0 (hash: 7b9bd461) both as the underlying floating-point LP solver in SCIP, and as an exact LP solver during our *a posteriori* analysis.

Preprocessing. Before passing the instances to SCIP and starting the actual branch-and-bound process, we apply simple presolving steps [2,4]. First, we perform some *model cleanup*, i.e., we delete already satisfied constraints and convert singleton rows to variable bounds. Further, we apply a limited form of *constraint propagation* in rational arithmetic in order to reduce the number of infinite bounds, which helps to increase the success rate of safe dual bounding.

Event system in SCIP. In order to track SCIP's branch-and-bound process, we register so-called events that inform us every time a leaf is reached. In detail, our implementation uses the following events. Every time SCIP finds a solution, it triggers the event BESTSOLUTION found. A NODEINFEASIBLE event is triggered both when the LP relaxation at this leaf is deemed infeasible, or when a node is pruned. The LP solving status can be accessed to differentiate whether the relaxed LP is infeasible or feasible and eventually pruned. Note that when SCIP creates new nodes it adds them to a queue and sets the local lower bound to the objective value of the LP relaxation of its parent. After a new solution is found, SCIP immediately removes all nodes from the queue whose lower bound is worse than the objective of the newly found solution. Each of these removed leaves triggers a NODEDELETE event. Finally, every integer-feasible node triggers an NODEFEASIBLE event.

All node events come with a list of branching decisions. Additionally for the NODEFEASIBLE and NODEINFEASIBLE events, also the primal and dual LP solution, respectively the Farkas proof [11], and the simplex basis are available. When the NODEDELETE event is triggered, we recompute the dual LP solution by solving the LP again in floating-point arithmetic with SoPlex. At the BESTSOLUTION event, SCIP only provides the primal solution.

Test sets. We consider two test sets curated in [6] for benchmarking an exact MIP solver. The FPEasy test set consists of 57 instances that were found easy by the floating-point solver SCIP. The NumDiff test set consists of 50 instances and was compiled to be numerically challenging due to large coefficient ranges or poor conditioning. For a detailed description we refer to [6], which also lists on which instances the results of floating-point SCIP deviate from the exact rational results.

To provide more stable results, we perform our experiments on each instance twice: one time on the original instance and another time after permuting the order of variables and constraints. For the NumDiff test set instance normalized-aim-200-1_6-ye we use aim-200 as an abbreviation.

The code base used for our experiments is publicly available at https://github.com/alexhoen/bnbanalyzer. All experiments were carried out on identical machines with Intel(R) Xeon(R) CPU E7-8880 v4 @ 2.20 GHz with a time limit of 10800 seconds for each instance, which includes both the solving time of the floating-point solver SCIP as well as the evaluation of the leaves.

3.2 Evaluation of branch-and-bound decisions in SCIP

Table 1 provides a first overview of the results for the floating-point solver SCIP on both test sets FPEasy and NumDiff, summarizing how many instances were correctly solved and on how many SCIP returns an inexact solution or even an incorrect status. The columns "correct" state the number of instances where no errors occurred as defined in Section 2.2: All solutions are either exactly integer feasible or convertible to such, all infeasible leaves are infeasible in rational arithmetic, and any node pruning is also justified in exact arithmetic. By contrast, the columns "fails" state the number of instances where at least one such error occurred and solving finished within the time limit. The column "primal" reports the number of instances with a solution error (see Section 2.2), the column "dual" reports the number of instances with a bound, gap, or infeasibility error.

Note that for two instances (30:70:4_5:0_95:100 and NPMV07) SCIP timed out before reaching a leaf or producing a solution and therefore did not issue any event. Since no wrong decision was made these instances are labeled as correct.

			within time limit				time out	
test set		instances	correct	fails	primal	dual	correct	fails
FPEasy	original	57	49	2	0	2	6	0
	permuted	57	49	2	0	2	6	0
NumDiff	original	50	11	20	12	9	14	5
	permuted	50	13	18	12	9	13	7

Table 1. Aggregate statistics on instances with/without incorrect decisions in floating-point SCIP.

Regardless of the permutation, for 55 of the 57 instances of FPEasy, we could verify that SCIP followed a fully correct solving process. On 49 instances it terminated with an exact optimal solution, while on 6 instances SCIP hit the time limit. Only for two instances, we encountered dual fails. On one instance (VPM2) the results is still correct, on the other instance (DANO3_4), the result is slightly suboptimal. This can also be seen in Table 5, where we compare the results of floating-point SCIP to the results of a numerically exact version of SCIP [6,8,9], on the instances where SCIP made at least one wrong decisions.

As expected, verifying the results of the NumDiff test set leads to more errors. Only on 25 (original) and 27 (permuted) of the 50 instances, respectively, SCIP followed a fully correct solution process. On each seed, 14 of these instances hit the time limit, so only for 11 respectively 13 instances SCIP terminated with a verifiable exact optimal solution. Still, on both test sets, a notable amount of the floating-point solves exhibits no errors.

For a more detailed analysis, Tables 2 to 4 include all instances labeled as "fail" in Table 1 and those instances where SCIP produced a "correct" result but made incorrect decisions during the solving process. All instances where SCIP made a wrong decision on FPEasy are listed in Table 2. Instances of NumDiff that encountered errors and are solved within the time limit are listed in Table 3. A complete overview NumDiff instances with wrong decisions where SCIP timed out are listed in Table 4. The column "leaves" lists the total number of processed leaves of the tree; the remaining columns report the number of leaves with errors according to the definition in Section 2.2, where "W" and "S" stands for weak and strong errors, respectively.

Instance	Perm	leaves	Sol W	Sol S	Bound W	Bound S	Gap W	Gap S
DANO3-4	0	29	0	0	0	0	0	1
	1	29	0	0	0	0	0	1
VPM2	0	688 018	0	0	5	34	0	0
	1	649 085	0	0	11	167	0	0

Table 2. Analysis of number of leaves with incorrect decisions on the FPEASY instances within the time limit.

Notably, only the instances NS1859355 and VPM2 show a correct result despite wrong decisions during the solving process. On FPEASY the vast majority of instances in Table 2 are solved without wrong decisions. SCIP incorrectly evaluated nodes on only two instances, affecting one node in DANO3-4 and 39 of 178 nodes in VPM2, respectively. Though 39 or 178 wrong decisions on VPM2 seems to be a notable amount of wrong decisions, compared to the overall evaluated nodes on the entire FPEASY test set the wrong decisions on these two instances make up a tiny portion.

As expected, on the numerical more challenging NUMDIFF test, more wrong decisions are made. First, let us focus on the instances with dual fails. Besides the instances ALU16_8, TKATTV5, and TKAT3TV, all instances show at most two leaves with strong bound or gap errors. On the instances, TKAT3*, NS1629327 and NS1859355, more weak and strong bound or gap errors appear, and ALU10_9 produces 46 infeasibility errors. Nevertheless, these all make up only a small fraction of the total number of evaluated nodes.

On the ALU*-instances (except for ALU16_1), and on the instances DFN6_LOAD, NEOS-1053591, NEOS-1062641, NEOS-1603965, NS1866531 the solutions generated by SCIP are slightly infeasible and can not be converted to exact solutions. Especially the ALU instances are numerical difficult: Either the instance is infeasible or there is a huge gap between the floating-point solution and the solution of EXACT-SCIP as can be seen in Table 5.

One natural attempt to reduce such solution errors is to call SCIP with a tightened feasibility tolerance. Per default, the feasibility tolerance is 10^{-6}. We repeated our experiments but tightened the tolerance to 10^{-9}. The results show a significant reduction in solution errors. On the instances ALU10_1, ALU10_8, ALU10_9, BERND2, AIM-200 and DFN6_LOAD, SCIP finds an exactly integer-feasible solution, or they can be converted to such, or SCIP times out, but on the instances 16_7 or NS1859355, the floating-point solver still produces slightly infeasible solutions. On the FPEASY instance DANO3_4 also all gap errors were removed, successfully closing the dual gap on this instance.

Instance	Perm	leaves	Sol W	Sol S	Bound W	Bound S	Gap W	Gap S	Inf
ALU10_1	0	883	2	0	0	0	0	0	0
ALU10_7	0	811	0	1	0	0	0	0	0
	1	734	0	3	0	0	0	0	0
ALU10_8	0	4 254	0	3	0	0	0	0	0
	1	4 486	1	3	0	0	0	0	0
ALU10_9	0	6 665	1	3	0	0	0	0	0
	1	7 675	1	5	0	0	0	0	0
ALU16_1	0	958 643	0	0	0	0	0	0	46
ALU16_7	0	694	0	3	0	0	0	0	0
	1	1 307	0	4	1	1	0	0	0
ALU16_8	0	TL	0	8	0	20	0	0	3
	1	219 362	0	3	0	2	0	0	0
ALU16_9	0	199 225	0	4	0	0	0	0	0
	1	44 961	0	13	0	0	0	0	0
BERND2	0	31 081	0	4	0	0	0	0	0
	1	23 144	0	5	0	0	0	0	0
DFN6_LOAD	0	11 210	2	6	9	0	0	0	0
	1	3 064	0	6	9	0	0	0	0
AIM-200	0	7 414	0	0	0	0	0	0	1
	1	9 534	0	0	0	0	0	0	1
NEOS-1053591	0	225 381	0	2	0	0	0	0	0
	1	TL	1	1	0	0	0	0	0
NEOS-1062641	0	92	0	1	0	0	0	0	0
	1	46	0	1	0	0	0	0	0
NEOS-1603965	0/1	1	0	1	0	0	0	0	0
NS1629327	0	9 985	0	0	0	0	0	2	0
	1	11 788	0	0	0	2	0	2	0
NS1859355	0	12 581	0	0	0	0	1	0	0
	1	6 434	0	0	0	0	2	0	0
NS1866531	0/1	1	0	1	0	0	0	0	0
PRODPLAN2	0	3	0	0	0	1	1	1	0
	1	27	0	0	3	0	2	1	0
TKAT3K	0	4 367	0	0	11	1	0	0	0
	1	8 335	0	0	5	2	0	0	0
TKAT3T	0	14 604	0	0	27	1	0	0	0
	1	5 219	0	0	4	4	0	0	0
TKAT3TV	0	4 546	0	0	10	7	0	0	0
	1	19 032	0	0	28	5	0	0	0
TKATTV5	0	8 459	0	0	15	26	0	0	0
	1	18 999	0	0	1	17	0	0	0

Table 3. Analysis of number of leaves with incorrect decisions on the NumDiff instances within the time limit.

Instance	Perm	leaves	Sol W	Sol S	Bound W	Bound S	Gap W	Gap S	Inf
alu16_5	0	1 342 439	0	0	0	0	0	0	11
	1	280 574	0	0	0	0	0	0	875
dfn6fp_load	1	585	0	2	3	4	0	0	0
ns2080781	0	701 865	7	3	156	72	0	0	0
	1	52 258	0	4	0	0	0	0	0
ns1925218	1	22 556	0	0	0	0	0	0	1
prodplan1	0	36	0	0	0	1	0	0	0
ran14x18-	0	1 268 265	0	12	1	0	0	0	0
-disj-8	1	1 288 277	0	6	2	2	0	0	0
neos-799716	1	1 095	0	0	0	0	0	0	1

Table 4. Analysis of number of leaves with incorrect decisions on the NumDiff instances with time out.

However, we also observe that the number of explored nodes is significantly increased. This does not only lead to increased solving time, but can also create more numerically challenging leaves, hence potentially increasing the absolute number of incorrect decisions. An example of this behavior is the instance blend2 where the result of SCIP can no longer be verified.

3.3 Analysis of different post-processing techniques

Finally, let us have a look at the effectiveness and success rate of the different post-processing techniques described in Section 2.3. Table 6 aggregates over all leaves of a complete run over each test set (column "leaves") and categorizes them based on the techniques that were successful for verification. The column "floating-point" lists the percentage of leaves that are verified automatically or through the use of safe bounding. The columns "Reconstruct", "Factorize", and "Exact" likewise represent the percentage for the corresponding methods. The row "unb" reports the numbers for all instances that contain variables with unbounded domains.

Surprisingly, 93.76% (FPEasy) to 99.69% (NumDiff) of the leaf decisions can be verified using floating-point arithmetic. Thus, the presented methods can be implemented very efficiently. In relative terms, the percentage of erroneous leaf decisions is small on all test sets.

Safe bounding requires bounded variables and can therefore more reliably be applied to problems with bounded variables. As a result, unbounded problems, listed in Table 6 in row "unb", have a significantly higher percentage of expensive Exact-SoPlex calls. On these unbounded problems, the success rate of the floating-point arithmetic drops significantly to 65.18% on FPEasy and 94.89%

Instance	SCIP Seed 0	SCIP Seed 1	range of exact solution
DANO3_4	576.435224722083	576.435224722083	576.4352247072
VPM2	13.75	13.75	13.75
ALU10_1	86	inf	inf
ALU10_7	83	83	[1243230.17385925,∞)
ALU10_8	84	84	[169.4062227788,∞)
ALU10_9	84	83	[1166.7499882579,3726773]
ALU16_1	84	inf	inf
ALU16_7	79	79	[315.4000244141,∞)
ALU16_8	x	79	[2256.0481863315,∞)
ALU16_9	79	79	[3801.0189752579,∞)
BERND2	11209.0573895658	11209.0573899187	113091.469015879
DFN6_LOAD	3.743842258432	3.743842258432	[3.7683622392,4.4007262645]
AIM-200	inf	inf	200
NEOS-1053591	-3662.9144	time limit	-3662.9144
NEOS-1062641	0	0	0
NEOS-1603965	619244367.662956	619244367.662956	[619246130.539662,∞)
NS1629327	-10.9803191329325	-10.9803191329325	-10.9803191329
NS1866531	0	0	10
PRODPLAN2	-239399.435140992	-239399.4351411	-239399.435141407
TKAT3K	4772818.1	4772818.1	4772818.1
TKAT3T	5564891.75	5564891.75	5564891.75
TKAT3TV	8388398.65	8388398.65	8388398.65
TKATTV5	28117644.225	28117644.225	28117644.225

Table 5. Comparison of the results of floating-point SCIP and rational solutions. The exact results were generated with EXACT-SCIP (hash: C7BD4BE7B9) and a time limit of 18800 seconds

		leaves	floating-point	Reconstruct	Factorize	EXACT	error
FPEASY	all	19952937	93.76%	2.93%	0.46%	2.85%	0.00020%
	unb	424881	65.18%	0.15%	4.78%	29.89%	0.00024%
NUMDIFF	all	8389838	99.69%	0.0%	0.04%	0.27%	0.00543515%
	unb	71221	94.89%	0.02%	1.22%	3.86%	0.00140408%

Table 6. Distribution on the different verification methods over all leaves on seed 0.

on NumDiff. This leads to an increase in more expensive techniques such as the Exact-SoPlex calls which significantly increase from 2.85% to 29.89% on FPEasy and from 0.06% to 3.86% on NumDiff.

4 Conclusion

Our experiments indicate that the overwhelming majority of the decisions made by the floating-point solver SCIP on both the test set FPEasy and even on the numerically more challenging test set NumDiff are correct even in rational arithmetic. For feasible instances, only a small, typically single-digit, number of nodes per instance can not be verified and would require further treatment to generate a proof of correctness of the floating-point calculation. Notably, most of the node decisions can be verified in fast floating-point arithmetic and do not require techniques that rely on more expensive rational arithmetic.

However, we also observe that sub-problems that are infeasible in exact arithmetic pose a special situation. Here, sometimes the floating-point solver can find solutions with slight violations and use them to prune nodes which are incorrect in terms of rational arithmetic. This is particularly troubling for instances that are globally infeasible in exact arithmetic. Our results show that this issue can only partially be addressed by tightening the feasibility tolerance of the solver.

Overall, these results are encouraging. On the one hand, they quantify the widely accepted belief that floating-point MIP solvers are generally numerically robust for well-behaved input data. On the other hand, they also suggest a path forward to using floating-point solvers as a grey box in order to solve MIPs exactly over the rational numbers. First, the experimental setup put forward in this paper can be used to generate partial certificates of optimality, which can even be verified for example in the VIPR format [5]. Second, these partial certificates can then be completed by continuing to explore sub-problems that were incorrectly discarded, either recursively with increased precision or by calling an exact MIP solver. For this to become competitive with the state of the art in exact MIP solving, our *a posteriori* verification of LP-based branch-and-bound needs to be extended to include more advanced techniques like presolving or cutting plane separation.

Acknowledgements. The work for this article has been partly conducted within the Research Campus MODAL funded by the German Federal Ministry of Education and Research (BMBF grant number 05M14ZAM).

References

1. Achterberg, T.: Constraint Integer Programming. Ph.D. thesis (01 2007). https://doi.org/10.14279/depositonce-1634
2. Achterberg, T., Bixby, R., Gu, Z., Rothberg, E., Weninger, D.: Presolve reductions in mixed integer programming. INFORMS Journal on Computing **32** (11 2019). https://doi.org/10.1287/ijoc.2018.0857

3. Bolusani, S., Besançon, M., Bestuzheva, K., Chmiela, A., Dionísio, J., Donkiewicz, T., van Doornmalen, J., Eifler, L., Ghannam, M., Gleixner, A., Graczyk, C., Halbig, K., Hedtke, I., Hoen, A., Hojny, C., van der Hulst, R., Kamp, D., Koch, T., Kofler, K., Lentz, J., Manns, J., Mexi, G., Mühmer, E., Pfetsch, M.E., Schlösser, F., Serrano, F., Shinano, Y., Turner, M., Vigerske, S., Weninger, D., Xu, L.: The scip optimization suite 9.0 (2024), https://arxiv.org/abs/2402.17702
4. Brearley, A.L., Mitra, G., Williams, H.P.: Analysis of mathematical programming problems prior to applying the simplex algorithm. Math. Program. **8**(1), 54–83 (Dec 1975). https://doi.org/10.1007/BF01580428, https://doi.org/10.1007/BF01580428
5. Cheung, K.K.H., Gleixner, A., Steffy, D.: Verifying integer programming results. In: F. Eisenbrand and J. Koenemann, eds., Integer Programming and Combinatorial Optimization: 19th International Conference, IPCO 2017. vol. 10328, pp. 148 – 160 (2017). https://doi.org/10.1007/978-3-319-59250-3_13
6. Cook, W., Koch, T., Steffy, D.E., Wolter, K.: An exact rational mixed-integer programming solver. In: Günlük, O., Woeginger, G.J. (eds.) Integer Programming and Combinatoral Optimization. pp. 104–116. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
7. Cook, W., Koch, T., Steffy, D.E., Wolter, K.: A hybrid branch-and-bound approach for exact rational mixed-integer programming. Mathematical Programming Computation **5**(3), 305–344 (2013). https://doi.org/10.1007/s12532-013-0055-6
8. Eifler, L., Gleixner, A.: A computational status update for exact rational mixed integer programming. Mathematical Programming (2022). https://doi.org/10.1007/s10107-021-01749-5
9. Eifler, L., Gleixner, A.: Safe and verified gomory mixed integer cuts in a rational mip framework (2023), https://arxiv.org/abs/2303.12365
10. Eifler, L., Gleixner, A., Pulaj, J.: A safe computational framework for integer programming applied to chvátal's conjecture. ACM Transactions on Mathematical Software **48**(2) (2022). https://doi.org/10.1145/3485630
11. Farkas, J.: Theorie der einfachen ungleichungen. Journal für die reine und angewandte Mathematik **124**, 1–27 (1902), http://eudml.org/doc/149129
12. Gleixner, A., Hendel, G., Gamrath, G., Achterberg, T., Bastubbe, M., Berthold, T., Christophel, P.M., Jarck, K., Koch, T., Linderoth, J., Lübbecke, M., Mittelmann, H., Ozyurt, D., Ralphs, T., Salvagnin, D., Shinano, Y.: Miplib 2017: Data-driven compilation of the 6th mixed-integer programming library. Mathematical Programming Computation **13**(3), 443 – 490 (2021). https://doi.org/10.1007/s12532-020-00194-3
13. Gleixner, A., Steffy, D.E.: Linear programming using limited-precision oracles. Mathematical Programming **183**(1–2), 525–554 (Nov 2019). https://doi.org/10.1007/s10107-019-01444-6, http://dx.doi.org/10.1007/s10107-019-01444-6
14. Gleixner, A.M., Steffy, D.E., Wolter, K.: Iterative refinement for linear programming. INFORMS J. on Computing **28**(3), 449–464 (Aug 2016)
15. Klotz, E.: Identification, assessment, and correction of ill-conditioning and numerical instability in linear and integer programs. In: Newman, A., Leung, J. (eds.) Bridging Data and Decisions, pp. 54–108. TutORials in Operations Research (2014). https://doi.org/10.1287/educ.2014.0130
16. Land, A.H., Doig, A.G.: An automatic method of solving discrete programming problems. Econometrica **28**(3), 497–520 (1960), http://www.jstor.org/stable/1910129

17. Neumaier, A., Shcherbina, O.: Safe bounds in linear and mixed-integer linear programming. Mathematical Programming **99**, 283–296 (2004). https://doi.org/10.1007/s10107-003-0433-3, https://doi.org/10.1007/s10107-003-0433-3
18. Pan, V.Y.: Nearly optimal solution of rational linear systems of equations with symbolic lifting and numerical initialization. Computers & Mathematics with Applications **62**(4), 1685–1706 (2011). https://doi.org/https://doi.org/10.1016/j.camwa.2011.06.006, https://www.sciencedirect.com/science/article/pii/S0898122111004743
19. Paxian, T., Biere, A.: Uncovering and classifying bugs in MaxSAT solvers through fuzzing and delta debugging. In: Järvisalo, M., Le Berre, D. (eds.) Proceedings of the 14th Internantional Workshop on Pragmatics of SAT Co-located with the 26th International Conference on Theory and Applicationas of Satisfiability Testing (SAT 2003), Alghero, Italy, July, 4, 2023. CEUR Workshop Proceedings, vol. 3545, pp. 59–71. CEUR-WS.org (2023), http://ceur-ws.org/Vol-3545/paper5.pdf
20. Saunders, B.D., Wood, D.H., Youse, B.S.: Numeric-symbolic exact rational linear system solver. In: Proceedings of the 36th International Symposium on Symbolic and Algebraic Computation. p. 305–312. ISSAC '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1993886.1993932, https://doi.org/10.1145/1993886.1993932
21. Steffy, D., Wolter, K.: Valid linear programming bounds for exact mixed-integer programming. Tech. Rep. 11-08, ZIB, Takustr. 7, 14195 Berlin (2011)
22. Steffy, D.E.: Topics in exact precision mathematical programming. Ph.D. thesis, Georgia Institute of Technology (2011), http://hdl.handle.net/1853/39639
23. Ursic, S., Patarra, C.: Exact solution of systems of linear equations with iterative methods. SIAM Journal on Algebraic Discrete Methods **4**(1), 111–115 (1983). https://doi.org/10.1137/0604014, https://doi.org/10.1137/0604014
24. Vries, S., Vohra, R.: Combinatorial auctions: A survey. INFORMS Journal on Computing **15**, 284–309 (08 2003). https://doi.org/10.1287/ijoc.15.3.284.16077
25. Wan, Z.: An algorithm to solve integer linear systems exactly using numerical methods. Journal of Symbolic Computation **41**(6), 621–632 (2006). https://doi.org/https://doi.org/10.1016/j.jsc.2005.11.001, https://www.sciencedirect.com/science/article/pii/S0747717105001653