Mathematical Programming (2023) 197:793–812
https://doi.org/10.1007/s10107-021-01749-5

FULL LENGTH PAPER

Series B

A computational status update for exact rational mixed integer programming

Leon Eifler[1] · Ambros Gleixner[1,2]

Received: 31 May 2021 / Accepted: 24 November 2021 / Published online: 7 January 2022

Abstract
The last milestone achievement for the roundoff-error-free solution of general mixed integer programs over the rational numbers was a hybrid-precision branch-and-bound algorithm published by Cook, Koch, Steffy, and Wolter in 2013. We describe a substantial revision and extension of this framework that integrates symbolic presolving, features an exact repair step for solutions from primal heuristics, employs a faster rational LP solver based on LP iterative refinement, and is able to produce independently verifiable certificates of optimality. We study the significantly improved performance and give insights into the computational behavior of the new algorithmic components. On the MIPLIB 2017 benchmark set, we observe an average speedup of 10.7x over the original framework and 2.9 times as many instances solved within a time limit of two hours.

Keywords Mixed integer programming · Exact computation · Rational arithmetic · Symbolic computations · Certificate of correctness

1 Introduction

It is widely accepted that mixed integer programming (MIP) is a powerful tool for solving a broad variety of challenging optimization problems and that state-of-the-art

The work for this article has been conducted within the Research Campus Modal funded by the German Federal Ministry of Education and Research (BMBF grant numbers 05M14ZAM, 05M20ZBM). An extended abstract of this article appeared in the 22nd conference on Integer Programming and Combinatorial Optimization (IPCO 2021) [18].

✉ Leon Eifler
eifler@zib.de

Ambros Gleixner
gleixner@zib.de

1 Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany

2 HTW Berlin, Treskowallee 8, 10313 Berlin, Germany

MIP solvers are sophisticated and complex computer programs. However, virtually all established solvers today rely on fast floating-point arithmetic. Hence, their theoretical promise of global optimality is compromised by roundoff errors inherent in this incomplete number system. Though tiny for each single arithmetic operation, these errors can accumulate and result in incorrect claims of optimality for suboptimal integer assignments, or even incorrect claims of infeasibility. Due to the nonconvexity of MIP, even performing an a posteriori analysis of such errors or postprocessing them becomes difficult.

In several applications, these numerical caveats can become actual limitations. This holds in particular when the solution of mixed integer programs is used as a tool in mathematics itself. Examples of recent work that employs MIP to investigate open mathematical questions include [11,12,19,29,30,33]. Some of these approaches are forced to rely on floating-point solvers because the availability, the flexibility, and most importantly the computational performance of MIP solvers with numerically rigorous guarantees is currently limited. This makes the results of these research efforts not as strong as they could be. Examples for industrial applications where the correctness of results is paramount include hardware verification [1] or compiler optimization [36].

The milestone paper by Cook, Koch, Steffy, and Wolter [16] presents a hybrid-precision branch-and-bound implementation that can still be considered the state of the art for solving general mixed integer programs exactly over the rational numbers. It combines symbolic and numeric computation and applies different dual bounding methods [20,32,34] based on linear programming (LP) in order to dynamically trade off their speed against robustness and quality.

However, beyond advanced strategies for branching and bounding, [16] does not include any of the supplementary techniques that are responsible for the strong performance of floating-point MIP solvers today. In this paper, we make a first step to address this research gap in two main directions.

First, we incorporate a *symbolic presolving* phase, which safely reduces the size and tightens the formulation of the instance to be passed to the branch-and-bound process. This is motivated by the fact that presolving has been identified by several authors as one of the components—if not *the* component—with the largest impact on the performance of floating-point MIP solvers [2,4]. To the best of our knowledge, this is the first time that the impact of symbolic preprocessing routines for general MIP is analyzed in the literature.

Second, we complement the existing dual bounding methods by enabling the use of *primal heuristics*. The motivation for this choice is less to reduce the total solving time, but rather to improve the usability of the exact MIP code in practical settings where finding good solutions earlier may be more relevant than proving optimality eventually. Similar to the dual bounding methods, we follow a hybrid-precision scheme. Primal heuristics are exclusively executed on the floating-point approximation of the rational input data. Whenever they produce a potentially improving solution, this solution is checked for approximate feasibility in floating-point arithmetic. If successful, the solution is postprocessed with an exact repair step that involves an exact LP solve.

Moreover, we integrate the exact LP solver SoPlex, which follows the recently developed scheme of *LP iterative refinement* [25], we extend the logging of *certificates* in the recently developed VIPR format to all available dual bounding methods [13], and produce a thoroughly *revised implementation* of the original framework [16], which improves multiple technical details. Our computational study evaluates the performance of the new algorithmic aspects in detail and indicates a significant overall speedup compared to the original framework.

The overarching goal and contribution of this research is to extend the computational practice of MIP to the level of rigor that has been achieved in recent years, for example, in the field of satisfiability solving [35], while at the same time retaining most of the computational power embedded in floating-point solvers. In MIP, a similar level of performance and rigor is certainly much more difficult to reach in practice, due to the numerical operations that are inherently involved in solving general mixed integer programs. However, we believe that there is no reason why this vision should be *fundamentally* out of reach for the rich machinery of MIP techniques developed over the last decades. The goal of this paper is to demonstrate the viability of this agenda within a first, small selection of methods. The resulting code is freely available for research purposes as an extension of SCIP 7.0 [17].

2 Numerically exact mixed integer programming

In the following, we describe related work in numerically exact optimization, including the main ideas and features of the framework that we build upon.

Before turning to the most general case, we would like to mention that roundoff-error-free methods are available for several specific classes of pure integer problems. One example for such a combinatorial optimization problem is the traveling salesman problem, for which the branch-and-cut solver Concorde applies safe interval-arithmetic to postprocess LP relaxation solutions and ensures the validity of domain-specific cutting planes by their combinatorial structure [5].

A broader class of such problems, on binary decision variables, is addressed in *satisfiability solving* (SAT) and *pseudo-Boolean optimization* (PBO) [10]. Solvers for these problem classes usually do not suffer from numerical errors and often support solver-independent verification of results [35]. While optimization variants exist, the development of these methods is to a large extent driven by feasibility problems. The broader class of solvers for *satisfiability modulo theories* (SMT), e.g., [31], may also include real-valued variables, in particular for satisfiability modulo the theory of linear arithmetic. However, as pointed out also in [21], the target applications of SMT solvers differ significantly from the motivating use cases in LP and MIP.

Exact optimization over convex polytopes intersected with lattices is also supported by some software libraries for polyhedral analysis [7,8]. These tools are not particularly targeted towards solving LPs or MIPs of larger scale and usually follow the naive approach of simply executing all operations symbolically, in exact rational arithmetic. This yields numerically exact results and can even be highly efficient as long as the size of problems or the encoding length of intermediate numbers is limited. However, as pointed out by [20] and [16], this *purely symbolic approach* quickly becomes prohibitively slow in general.

By contrast, the most effective methods in the literature rely on a *hybrid approach* and combine exact and numeric computation. For solving pure LPs exactly, the most recent methods that follow this paradigm are *incremental precision boosting* [6] and *LP iterative refinement* [25]. In an exact MIP solver, however, it is not always necessary to solve LP relaxations completely, but it often suffices to provide dual bounds that underestimate the optimal relaxation value safely. This can be achieved by postprocessing approximate LP solutions. *Bound-shift* [32] is such a method that only relies on directed rounding and interval arithmetic and is therefore very fast. However, as the name suggests it requires upper and lower bounds on all variables in order to be applicable. A more widely applicable bounding method is *project-and-shift* [34], which uses an interior point or ray of the dual LP. These need to be computed by solving an auxiliary LP exactly in advance, though only once per MIP solve. Subsequently, approximate dual LP solutions can be corrected by projecting them to the feasible region defined by the dual constraints and shifting the result to satisfy sign constraints on the dual multipliers.

The hybrid branch-and-bound method of [16] combines such safe dual bounding methods with a state-of-the-art branching heuristic, reliability branching [3]. It maintains both the exact problem formulation

$$\min\{c^T x \mid Ax \geq b,\ x \in \mathbb{Q}^n,\ x_i \in \mathbb{Z}\ \forall i \in \mathcal{I}\}$$

with rational input data $A \in \mathbb{Q}^{m\times n}, c \in Q^n, b \in \mathbb{Q}^m$, as well as a floating-point approximation with data $\bar{A}, \bar{b}, \bar{c}$, which are defined as the componentwise closest numbers representable in floating-point arithmetic. The set $\mathcal{I} \subseteq \{1, \ldots, n\}$ contains the indices of integer variables.

During the solve, for all LP relaxations, the floating-point approximation is first solved in floating-point arithmetic as an approximation and then postprocessed to generate a valid dual bound. The methods available for this safe bounding step are the previously described *bound-shift* [32], *project-and-shift* [34], and an *exact LP solve* with the exact LP solver QSOPT_EX based on incremental precision boosting [6]. (Further dual bounding methods were tested, but reported as less important in [16].) On the primal side, all solutions are checked for feasibility in exact arithmetic before being accepted.

Finally, this exact MIP framework was recently extended by the possibility to generate a *certificate* of correctness [13]. This certificate is a tree-less encoding of the branch-and-bound search, with a set of dual multipliers to prove the dual bound at each node or its infeasibility. Its correctness can be verified independently of the solving process using the checker software VIPR [14].

3 Extending and improving an exact MIP framework

The exact MIP solver presented here extends [16] in four ways: the addition of a symbolic presolving phase, the execution of primal floating-point heuristics coupled with an exact repair step, the use of a recently developed exact LP solver based on LP iterative refinement, and a generally improved integration of the exact solving routines into the core branch-and-bound algorithm.

3.1 Symbolic presolving

The first major extension is the addition of symbolic presolving. To this end, we integrate the newly available presolving library PAPILO [23] for integer and linear programming. PAPILO has several benefits for our purposes.

First, its code base is by design fully templatized with respect to the arithmetic type. This enables us to integrate it with rational numbers as data type for storing the MIP data and all its computations. Second, it provides a large range of presolving techniques already implemented. The ones used in our exact framework are coefficient strengthening, constraint propagation, implicit integer detection, singleton column detection, substitution of variables, simplification of inequalities, parallel row detection, sparsification, probing, dual fixing, dual inference, singleton stuffing, and dominated column detection. For a detailed explanation of these methods, we refer to [2]. Third, PAPILO comes with a sophisticated parallelization scheme that may help to compensate for the increased overhead introduced by the use of rational arithmetic. For details see [22].

When SCIP enters the presolving stage, we pass a rational copy of the problem to PAPILO, which executes its presolving routines iteratively until no sufficiently large reductions are found. Subsequently, we extract the postsolving information provided by PAPILO to transfer the model reductions to SCIP. These include fixings, aggregations, and bound changes of variables and strengthening or deletion of constraints, all of which are performed in rational arithmetic.

3.2 Primal heuristics

The second extension is the safe activation of SCIP's floating-point heuristics and the addition of an exact repair heuristic for their approximate solutions. Heuristics are not known to reduce the overall solving time drastically, but they can be particularly useful on hard instances that cannot be solved at all, and in order to avoid terminating without a feasible solution.

In general, activating SCIP's floating-point heuristics does not interfere with the exactness of the solving process, although care has to be taken that no changes to the model are performed, e.g., the creation of a no-good constraint. However, the chance that these heuristics find a solution that is feasible in the exact sense can be low, especially if equality constraints are present in the model. Thus, we postprocess solutions found by floating-point heuristics in the following way. First, we fix all integer variables to the values found by the floating-point heuristic, rounding slightly fractional values to their nearest integer. Then an exact LP is solved for the remaining continuous subproblem. If that LP is feasible, this produces an exactly feasible solution to the mixed integer program.

Clearly, these calls to the exact LP solver can create a significant overhead compared to executing a floating-point heuristic alone. This holds especially when a large percentage of the variables is continuous and thus cannot be fixed.

We take three steps to mitigate this overhead. First, we only attempt to repair solutions whose floating-point objective value is better than the incumbent. Second, we only process those solutions directly that improve on the imcumbent by more than

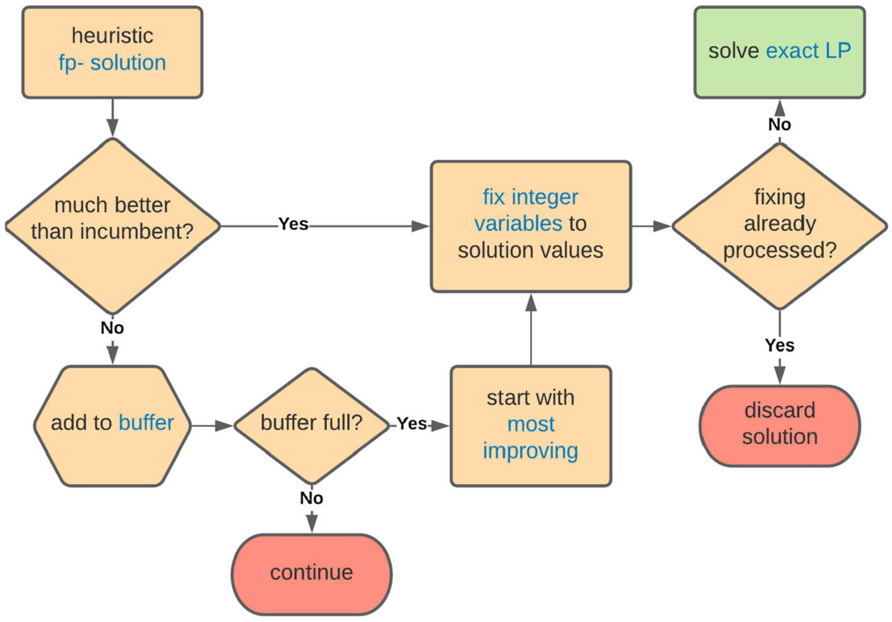
heuristic fp- solution
much better than incumbent?
Yes
No
fix integer variables to solution values
fixing already processed?
No
solve exact LP
Yes
discard solution
add to buffer
buffer full?
Yes
No
start with most improving
continue

Fig. 1 Schematic of exact repair step with buffering and hashing of previously processed solutions

20%. All other found solutions are buffered and only processed in increasing order of objective value once the buffer is full.

Third, we noticed that heuristics frequently find solutions that are either identical or differ only in the continuous solution values, but share the same integer variable assignments. Therefore, we store all previously prcossed partial integer solutions in a hash table and discard solutions that were already processed before. This exact repair algorithm is visualized in Fig. 1.

3.3 LP iterative refinement

Exact linear programming is a crucial part of the exact MIP solving process. Instead of QSOPT_EX, we use SOPLEX as the exact linear programming solver. The reason for this change is that SOPLEX uses LP iterative refinement [26] as the strategy to solve LPs exactly, which compares favorably against incremental precision boosting for solving pure LPs from scratch [25].

A speedup factor of 5.5 was reported [25] on linear programming instances where refinement was necessary, i.e., where the unrefined floating-point optimal basis was not the exact optimal basis. However, it is open how this result translates to the setting in an exact MIP solver, for several reasons. First, we expect that the final basis of the initial floating-point solve is already optimal for the majority of LPs encountered during our MIP solves. Second, most LP relaxations are solved from a near-optimal starting basis.

Furthermore, LP iterative refinement has the disadvantage that it can break down when the LP is too ill-conditioned to be solved by the underlying floating-point solver. While QSOPT_EX can incrementally boost the precision of its simplex routines, even to a fully rational solve if necessary, the current implementation of LP iterative refinement in SOPLEX relies on double-precision simplex solves. This may affect the performance of the exact MIP solver on numerically difficult instances. Although this rarely happens in practice, there exist instances where we observed this effect. It is further discussed in Sect. 4.3.

3.4 Further enhancements

We improved several algorithmic details in the implementation of the hybrid branch-and-bound method. We would like to highlight two examples for these changes. First, we enable the use of an *objective limit* in the floating-point LP solver, which was not possible in the original framework. Passing the primal bound as an objective limit to the floating-point LP solver allows the LP solver to stop early just after its dual bound exceeds the global primal bound. However, if the overlap is too small, postprocessing this LP solution with safe bounding methods can easily lead to a dual bound that no longer exceeds the objective limit. For this reason, before installing the primal bound as an objective limit in the LP solver, we increase it by the average amount of increase that was observed in the safe dual bounding step. To avoid problems at the start of the solve, where this average is not yet well-established, we also set a minimum value of 10^{-6} for the increase. Only when safe dual bounding fails, the floating-point LP is solved again without objective limit.

Second, we reduce the time needed for checking exact feasibility of primal solutions by prepending a safe floating-point check. Although checking a single solution for feasibility is fast, this happens often throughout the solve and doing so repeatedly in exact arithmetic can become computationally expensive. To implement such a safe floating-point check, we employ *running error analysis* [28]. Let $x^* \in \mathbb{Q}^n$ be a potential solution and let $\bar{x}^*$ be the floating-point approximation of x^*. Let $a \in \mathbb{Q}^n$ be a row of A with floating-point approximation $\bar{a}$, and right hand side $b_j \in \mathbb{Q}$. Instead of computing $\sum_{i=1}^n a_i x_i^*$ symbolically, we instead compute $\sum_{i=1}^n \bar{a}_i \bar{x}_i^*$ in floating-point arithmetic, and alongside compute a bound on the maximal rounding error that may occur. We adjust the running error analysis described in [28, Alg. 3.2] to also account for roundoff errors $|\bar{x}^* - x^*|$ and $|\bar{a} - a|$. The resulting method is shown in Algorithm 1.

After performing this computation, we can check if either $s - \mu \geq b_j$ or $s + \mu \leq b_j$. In the former, the solution x^* is guaranteed to fulfill $\sum_{i=1}^n a_i x_i^* \geq b_j$; in the latter, we can safely determine that the inequality is violated; only if neither case occurs, we recompute the activity in exact arithmetic.

We note that this could alternatively be achieved by directed rounding, which would give tighter error bounds at a slightly increased computational effort. However, empirically we have observed that most equality or inequality constraints are either satisfied at equality, where an exact arithmetic check cannot be avoided, or they are violated or satisfied by a slack larger than the error bound μ, hence the running error analysis is sufficient to determine feasibility.

Algorithm 1 Running error analysis applied to activity computation

```
1: procedure ErrorAnalysis($\bar{a}$, $\bar{x}$, $\delta$)
2:     Input: $\bar{x}^*, \bar{a} \in \mathbb{Q}^n$, maximal unit roundoff $\delta \in \mathbb{R}$
3:     Output $s, y \in \mathbb{R}$, with $|a^T x^* - s| \leq \mu$
4:     $s = 0$
5:     for $i = 1, \dots, n$ do
6:         $s \leftarrow s + \bar{a}_i \bar{x}_i^*$
7:         $\mu \leftarrow \mu + |s| + (3 + \delta)|\bar{a}_i \bar{x}_i^*|$
8:     end for
9:     $\mu \leftarrow \mu\delta$
10: end procedure
```

4 Computational study

We conduct a computational analysis to answer four main questions. *First, how does the revised branch-and-bound framework compare to the previous implementation, and to which components can the changes be attributed?* To answer this question, we compare the original framework [16] against our improved implementation, but with primal heuristics and exact presolving still disabled. As a first step, we use QSopt_ex as the exact LP solver for both frameworks. In particular, we analyze the importance and performance of the different dual bounding methods. Then, we evaluate the impact of LP iterative refinement by comparing our new implementation against itself, once with SoPlex as the exact LP solver and once with QSopt_ex.

Second, what is the impact of the new algorithmic components symbolic presolving and primal heuristics? To answer this question, we compare their impact on the solving time and the number of solved instances, as well as present more in-depth statistics, such as e.g., the primal integral [9] for heuristics or the number of fixings for presolving. In addition, we compare the effectiveness of performing presolving in rational and in floating-point arithmetic.

Third, what is the overhead for producing and verifying certificates? Here, we consider running times for both the solver and the certificate checker, as well as the overhead in the safe dual bounding methods introduced through enabling certificates. This provides an update for the analysis in [13], which was limited to the two bounding methods project-and-shift and exact LP.

Finally, how much is the overall improvement and how does it compare to the performance of floating-point SCIP? For this final question, we compare the overall performance of the new framework with all new components enabled against the original one. We also compare our framework against two versions of floating-point SCIP: the default, running all available SCIP features, and a reduced version, which only uses the features currently available for exact SCIP.

4.1 Computational setup and test sets

The experiments were performed on a cluster of Intel Xeon Gold 5122 CPUs with 3.6 GHz and 96 GB main memory. As in [16], we use CPLEX 12.3.0 as floating-point LP solver for all experiments. For exact LP solving, we use either the same QSOPT_EX version as in [16] or SOPLEX 5.0.2. For all symbolic computations, we use the GNU Multiple Precision Library (GMP) 6.1.4 [27]. For symbolic presolving, we use PAPILO 1.0.1 [22,23]; all other SCIP presolvers are disabled.

We consider three test sets in total. First, we use the two test sets specifically curated in [16]: one set with 57 instances that were found to be easy for an inexact floating-point branch-and-bound solver (FPEASY), and one set of 50 instances that were found to be numerically challenging, e.g., due to poor conditioning or large coefficient ranges (NUMDIFF). For a detailed description of the selection criteria, we refer to [16]. To complement these test sets with a set of more ambitious and recent instances, we also run all tests on the subset of MIPLIB 2017 [24] benchmark instances that could be solved to optimality by SCIP 7.0.2 within two hours.

All experiments to evaluate the new code are run with three different random seeds, where we treat each instance-seed combination as a single observation. As this feature is not available in the original framework, all comparisons with the original framework were performed with one seed. The time limit was set to 7200 seconds for all experiments. If not stated otherwise all aggregated numbers are shifted geometric means with a shift of 0.001 seconds or 100 branch-and-bound nodes, respectively.

4.2 The branch-and-bound framework

As a first step, we compare the behavior of the safe branch-and-bound implementation from [16] with QSOPT_EX as the exact LP solver, against its revised implementation, also with QSOPT_EX as exact LP solver. Note that the original implementation is based on SCIP 3.0 and our revised one is based on SCIP 7.0. However, there are no significant changes in performance just based on the different SCIP versions. For this comparison, all new and improved features that make SCIP 7.0 faster are disabled, and the relevant core of SCIP has remained functionally unchanged between these two versions.

The original framework uses the "Auto-Ileaved" bounding strategy as recommended in [16]. It dynamically chooses the dual bounding method, attempting to employ bound-shift as often as possible. An exact LP is solved whenever a node would be cut

Table 1 Comparison of original and new framework with presolving and primal heuristics disabled

		Original framework				New framework			
Test set	Size	Solved	Time	Nodes	Dbtime	Solved	Time	Nodes	Dbtime
FPEASY	54	46	91.6	8983.9	59.2	53	65.5	6742.1	39.6
NUMDIFF	21	13	197.8	9761.9	136.8	19	204.1	10590.5	128.5
MIPLIB	35	18	2090.1	18089.1	1640.1	33	1220.9	12078.8	722.8

off within tolerances, but not with the computed exact safe dual bound. In the new implementation, we use a similar strategy, however, we additionally solve the exact LP every 4, 8, 16, . . . depth levels of the tree. This change is motivated by the improved performance in the exact LP solver when switching to SoPlex.

Table 1 reports the results for solving time, number of nodes, and total time spent in safe dual bounding (“dbtime”), for all instances that could be solved by at least one solver. The new framework could solve 7 instances more on FPEASY, 6 more on NUMDIFF, and 15 more on MIPLIB. On FPEASY and MIPLIB we observe a reduction in solving time (28.5% and 41.6%), while on NUMDIFF solving time increases by 3.2%. This is consistent with the number of solved nodes. On FPEASY and MIPLIB the number decreases by 25%, and 33.2%, respectively, while it increases by 8.5% on NUMDIFF. This means that our changes led to a significant reduction of the number of nodes, except on the numerically difficult test set, and a slight reduction in the node processing time across all test sets. In terms of the time spent in the safe dual bounding methods, we observe reductions of 33.1% (FPEASY), 6.1% (numdiff), and 56% (miplib).We also see this significant performance improvement reflected in the performance profiles in Fig. 2.

We identify a more aggressive use of project-and-shift, as well as more frequent exact LP solves as the two key factors for this improvement. In the original framework, project-and-shift is restricted to instances that had fewer than 10000 nonzeros. One

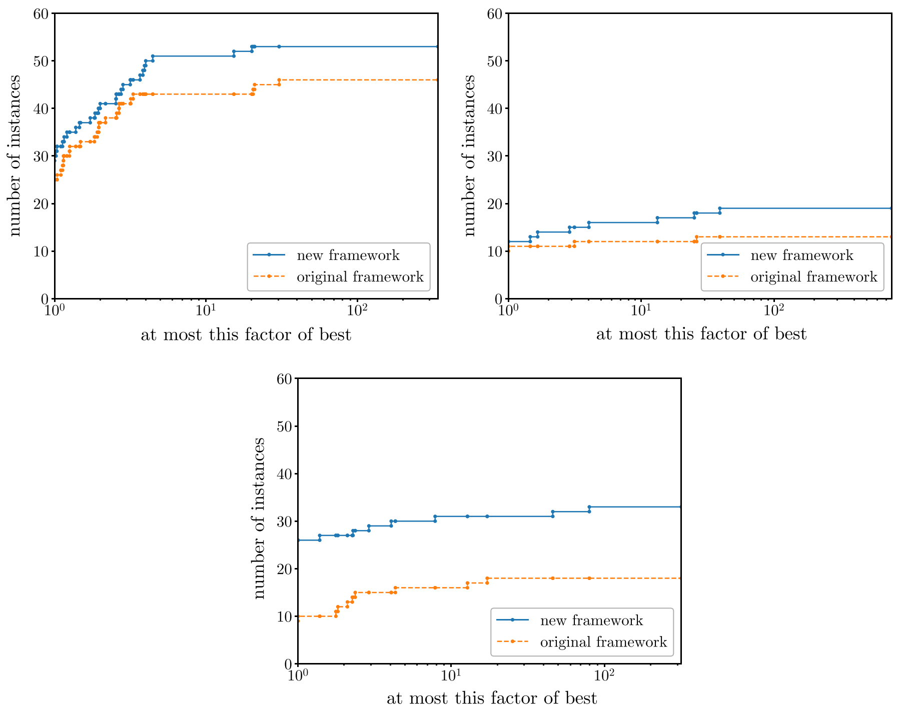

Fig. 2 Performance profiles comparing solving time of original and new framework without presolving and heuristics for FPEASY (top left), NUMDIFF (top right), MIPLIB (bottom)

Table 2 Comparison of safe dual bounding techniques

		Original framework			New framework		
Test set	Stats	bshift	pshift	exlp	bshift	pshift	exlp
FPEASY	Calls/node	1.00	0.83	0.34	0.66	0.84	0.20
	Time/call [s]	0.0016	0.0011	0.0416	0.0072	0.0017	0.0484
	Time/solving time	1.7 %	68.8 %	38.8 %	7.3 %	44.0 %	36.8 %
NUMDIFF	Calls/node	1.00	0.71	0.48	0.78	0.41	0.25
	Time/call [s]	0.0010	0.0028	0.1444	0.0039	0.0036	0.1936
	Time/solving time	1.1 %	50.0 %	60.3 %	3.8 %	13.9 %	60.0 %
MIPLIB	Calls/node	0.99	0.82	0.70	0.74	0.75	0.16
	Time/call [s]	0.0074	0.0029	0.2712	0.0276	0.0053	0.3240
	Time/solving time	1.4 %	79.4 %	70.1 %	15.6 %	47.4 %	33.4 %

reason for this limit is that a large auxiliary LP has to be solved by the exact LP solver to compute the relative interior point in project-and-shift. Our experiments showed that it was beneficial to remove this restriction, even when using QSOPT_EX as the exact LP solver.[1] This benefit only becomes more pronounced with improved exact LP solving performance. Of the 6 gained instances on NUMDIFF, 4 are subject to this restriction in the old framework. On MIPLIB, it is the case for 9 instances not solved by the original framework.

The reason we identify our more frequent exact LP solving as the second key factor, at least on FPEASY and MIPLIB, is that the improvement in solving time is mostly consistent with the reduction in the number of nodes. The only change in the general framework that we expect to systematically decrease the number of nodes is the interleaving of exact LP solving calls. On the other hand, average node-processing times do not change much when comparing the two frameworks. It is almost the same on FPEASY and NUMDIFF, while on MIPLIB the new framework has a slightly lower average time spent per node (0.101 instead of 0.115). This indicates that our other changes were effective in reducing the node-processing time.

A more detailed analysis of bounding times is given in Table 2. For a fair comparison, we always consider for each bounding method the subset of instances where both solvers ran the respective bounding method. For calls per node and the fraction of bounding time per total solving time, which are normalized well, we report the arithmetic means; for time per call, we report geometric means. Since the only fair comparison is to look at the subset of instances where both variants ran the same bounding methods, we cannot directly see the impact of more aggressive project-and-shift use here. We can however observe the impact of the problem in the original framework observed in footnote 1. Both the higher percentage of solving time spent in project-and-shift ("pshift") as well as the lower percentage spent in bound-shift ("bshift") is partly due to the unnecessary project-and-shift calls in the original framework.

[1] The reason this working limit was introduced may be a performance bug in the original framework that leads to project-and-shift being called even when bound-shift was successful. A portion of the improvement in solving time is most likely due to these previously unnecessary bounding calls.

The increase in bound-shift time per call is due to implementation details, that will be addressed in future versions of the code, but its fraction of the total solving time is still relatively low. One reason for the reduction in the number of bound-shift calls per node is that we now disable bound-shift dynamically if its success rate drops below 20%, while the original framework always ran bound-shift.

In terms of exact LP solving, we observe that the time per call increases slightly in the new framework, while the number of calls per node is reduced significantly. Both of these effects can be explained by the more aggressive project-and-shift usage. In the old framework, the exact LP is run very frequently on large instances where project-and-shift is disabled. In the new one on the other hand, the exact LP is only solved from time to time, either when project-and-shift fails or we are at a depth-level with an interleaving exact LP call. This also explains why the time-per-call is higher in the new framework: The exact LPs that are solved differ more from each other, making more iterations necessary to reach optimality. At the same time, disabling project-and-shift completely on larger instances leads to a large portion of solving time being spent in exact LP calls in the old framework, especially on MIPLIB, where larger instances are more frequent.

4.3 LP iterative refinement

In this section, we report on the performance impact of switching from QSOPT_EX to SOPLEX within the new framework, and therefore from incremental precision boosting to LP iterative refinement. Table 3 shows the overall performance comparison for instances that could be solved to optimality by at least one solver. We observe 3 more instances solved on FPEASY, 6 fewer on NUMDIFF, and 18 more on MIPLIB. It is noteworthy that all the instances solved by QSOPT_EX on which the SOPLEX-variant timed out are chip-verification instances [1], where using QSOPT_EX leads to drastically smaller search trees. As pointed out in Sect. 3.3, LP iterative refinement can fail to solve the exact LP completely in very ill-conditioned cases. This is the case for some LPs on these instances, forcing the version running SOPLEX to branch on any unfixed integer variable with the highest branching priority. This unsophisticated choice of both the variable as well as the branching value leads to these large search trees.

In terms of solving time, we observe a reduction of 59.2% on FPEASY, 37.7% on NUMDIFF, and of 60.1% on MIPLIB instances. As expected, the vast majority of this

Table 3 Comparison of employing precision boosting (QSOPT_EX) versus LP iterative refinement (SOPLEX)

		QSOPT_EX				SOPLEX			
Test set	Size	Solved	Time	Nodes	Dbtime	Solved	Time	Nodes	Dbtime
FPEASY	162	159	65.4	6741.8	39.7	162	26.8	5079.5	11.1
NUMDIFF	66	63	229.7	6000.9	146.6	57	143.1	5826.3	61.8
MIPLIB	132	108	1283.1	10879.2	801.4	126	512.3	9246.9	223.8

Table 4 Comparison of exact LP times with different exact LP solvers

	QSOPT_EX		SOPLEX	
Test set	Time/call [s]	Time/solving time (%)	Time/call [s]	Time/solving time (%)
FPEASY	0.0298	39.4	0.0071	20.1
NUMDIFF	0.2593	58.4	0.1021	43.6
MIPLIB	0.3222	33.3	0.1116	21.7

reduction comes from the reduced time spent in safe dual bounding methods, with the number of solved nodes being reduced only slightly: 24.7% on FPEASY, 3% on NUMDIFF, and 15% on MIPLIB.

Table 4 shows the percentage of total solving time spent in exact LP solving calls in arithmetic mean, as well as the geometric mean of time per exact LP solving call on the subset of instances where both variants solved LPs exactly. The time per exact LP call is reduced by 76.2% onFPEASY, 60.6% on NUMDIFF, and 65.4% on MIPLIB. This leads to a reduction of time spent in exact LP solving by 49% on FPEASY, 25.3% on NUMDIFF, and 34.8% on MIPLIB.

4.4 Symbolic presolving

Before measuring the overall performance impact of exact presolving, we address the question how effective and how expensive presolving in rational arithmetic is compared to standard floating-point presolving. For both variants, we configured PAPILO to use the same tolerances for determining whether a reduction found is strong enough to be accepted. The only difference in the rational version is that all computations are done in exact arithmetic and the tolerance to compare numbers and the feasibility tolerance are zero. Note that a priori it is unclear whether rational presolving yields more or fewer reductions. Dominated column detection may be less successful due to the stricter comparison of coefficients; the dual inference presolver might be more successful if it detects earlier that a dual multiplier is strictly bounded away from zero.

Table 5 presents aggregated results for presolving time, the number of presolving rounds, and the number of found fixings, aggregations, and bound changes. We use a shift of 1 for the geometric means of rounds, aggregations, fixings, and bound changes to account for instances where presolving found no such reductions. Remarkably, both variants yield virtually the same results on FPEASY. On NUMDIFF, there are small differences, with a slight decrease in the number of fixings and aggregations and a slight increase in the number of bound changes for exact variant. The time spent for exact presolving increases by more than an order of magnitude but symbolic presolving is still not a performance bottleneck. It consumed only 0.7% (FPEASY), 1.3% (NUMDIFF), and 0.6% (MIPLIB) of the total solving time, as seen in Table 6. Exploiting parallelism in presolving provided no measureable benefit for floating-point presolving, but reduced symbolic presolving time by 38.9% (FPEASY) to 43.7% (MIPLIB). However, this benefit can be more pronounced on individual instances, e.g., on `ex09` (MIPLIB), where parallelization reduces the time for rational presolving by a factor of 3.6 from 698 to 195 seconds.

Table 5 Comparison of exact and floating-point presolving

		Floating-point presolving					Exact presolving				
Test set	Thrds	Time	Rnds	Fixed	Agg	Bdchg	Time	Rnds	Fixed	Agg	Bdchg
FPEASY	1	0.01	3.2	8.8	3.6	10.3	0.18	3.2	8.8	3.6	10.3
	8	0.01	3.2	8.8	3.6	10.3	0.11	3.2	8.8	3.6	10.3
NUMDIFF	1	0.01	9.0	46.8	52.2	43.3	0.49	7.6	35.5	40.0	45.4
	8	0.03	9.0	46.8	52.2	43.2	0.34	7.6	35.5	40.0	45.4
MIPLIB	1	0.07	4.7	57.7	12.7	16.3	2.24	4.7	56.3	12.1	16.4
	8	0.11	4.7	57.7	12.7	16.3	1.26	4.7	56.3	12.1	16.4

Table 6 Comparison of new framework with and without presolving (3 seeds)

		Presolving disabled			Presolving enabled			
Test set	Size	Solved	Time	Nodes	Solved	Time	(Presolving)	Nodes
FPEASY	168	162	32.7	5594.7	168	21.0	(0.15)	4575.5
NUMDIFF	83	54	331.4	9948.7	78	52.7	(0.66)	2388.5
MIPLIB	145	123	667.7	11675.4	136	384.1	(2.40)	10594.4

To evaluate the impact of exact presolving, we compare the performance of the basic branch-and-bound algorithm established above against the performance with presolving enabled. The results for all instances that could be solved to optimality by at least one setting are presented in Table 6. Enabling presolving solves 6 more instances on FPEASY, 24 more instances on NUMDIFF, and 13 more instances on MIPLIB. We observe a reduction in solving time of 35.8% (FPEASY), 84.1% (NUMDIFF), and 42.4% (MIPLIB). The stronger impact on NUMDIFF is correlated with the larger number of reductions observed in Table 5.

4.5 Primal heuristics

To improve primal performance, we enabled all SCIP heuristics that the floating-point version executes by default. The repair heuristic is disabled on instances with more than 80% continuous variables since the overhead of the exact LP solves can drastically worsen the performance on those instances. We also set a limit of 1000 on the number of consecutive times the repair step is allowed to fail to produce a feasible solution. This limit is useful on instances that allow solutions within tolerances but are infeasible in exact arithmetic.

First, we evaluate the cost and success of the exact repair heuristic over all instances where it was called at least once. The results are presented in Table 7. The repair heuristic is effective at finding feasible solutions with a success rate of 57.5% (FPEASY), 47.3 (NUMDIFF), and 65.3% (MIPLIB). The lower success rate on NUMDIFF, meaning that the integer assignments found by the floating-point heuristic can less often be confirmed feasible, is expected since the test set by design contains instances where

Table 7 Statistics of repair heuristic for instances where repair step was called

		Time				
Test set	Size	Total solving	Repair	Fail	Success	Success rate
FPEASY	110	26.5	0.0174	0.0062	0.0051	57.6%
NUMDIFF	48	375.1	0.2343	0.0898	0.0833	47.3%
MIPLIB	218	2473.3	1.3155	0.4944	0.1691	65.3%

floating-point routines break down more easily. The fraction of the solving time spent in the repair heuristic is well below 1%. Nevertheless, the strict working limits we imposed are necessary since there exist outliers for which the repair heuristic takes more than 5% of the total solving time. Performance on these instances would quickly deteriorate if the working limits were relaxed.

Table 8 shows the overall performance impact of enabling heuristics over all instances that could be solved to optimality by at least one variant. Besides the solving time and the number of solved instances, also the time to find the first solution, as well as the primal integral [9] are reported as established measures of primal performance. Enabling heuristics solves 5 more instances on NUMDIFF and 6 more on MIPLIB. On FPEASY, the number of solved instances is unchanged. On FPEASY and MIPLIB we see only slight differences in total solving time. On NUMDIFF, solving time decreases by 18.9% when enabling heuristics. On FPEASY, the time to find the first solution decreases by 79.7% and the primal integral decreases by 27.8%. On NUMDIFF, time-to-first is reduced by 58.4% and the primal integral by 35.5%. The benefit is greatest on MIPLIB, with a reduction of 97.2% for time-to-first and of 62.4% in the primal integral.

We have to attribute a portion of the speedup on NUMDIFF to performance variability since some of the instances where the variant with enabled heuristics is much faster are two infeasible instances.[2] If we exclude those infeasible instances, the reduction in solving time is only 13%.

In all test sets, the repair heuristic was able to find solutions and improve the solving process on the primal side, while not imposing any significant overhead in solving time.

4.6 Producing and verifying certificates

The possibility to log certificates as presented in [13] is available in the new framework and is extended to also work when the dual bounding method bound-shift is active. Presolving must currently be disabled, since PAPILO does not yet support generation of certificates.

[2] In general, enabling primal heuristics should not improve the solving time on infeasible instances. However, any change in the algorithm can lead to different paths in the search tree, which in turn can drastically affect performance. The effect here is actually caused by the instances alu10_1 and alu16_2. We have rerun the experiment on both instances with 10 random seeds, and while the effect mostly disappears on alu16_2, the version with heuristics enabled is consistently faster on alu10_1. Unfortunately, such phenomena can occur since randomization is not perfect: many of the decisions inside the solver are still fully deterministic and thus not affected by a change in the random seed.

Table 8 Comparison of new framework with and without primal heuristics (3 seeds, presolving enabled, instances where repair step was called)

		Heuristics disabled			Heuristics enabled		
Test set	Size	Solv.time	Time-to-first	Primal int.	Solv.time	Time-to-first	Primal int.
FPEASY	110	25.7	0.64	173.0	26.5	0.13	125.7
NUMDIFF	48	462.6	23.04	3303.8	375.1	9.58	2129.3
MIPLIB	218	2519.7	96.47	29493.7	2473.3	2.67	11061.7

Table 9 Overhead for producing and verifying certificates on instances solved by both variants

		Certificate disabled		Certificate enabled			
Test set	Size	Solving time	Dbtime	Solving time	Dbtime	Check time	Overhead (%)
FPEASY	54	26.8	11.1	41.9	11.4	9.7	92.5
NUMDIFF	17	46.3	14.2	56.5	14.6	4.0	32.8
MIPLIB	39	378.7	155.2	552.3	165.6	75.4	65.8

Besides ensuring correctness of results, certificate generation is valuable to ensure correctness of the solver. Although it does not check the implementation itself, it can help identify and eliminate incorrect results that do not directly lead to wrong results. For example, on instance x_4 from NUMDIFF, the original framework claimed infeasibility at the root node. While the instance is indeed infeasible, we found the reasoning for this to be incorrect due to the use of a certificate. The issue was that a lower bound that exceeded the solvers default infinity value was found, and thus the root node was claimed to be infeasible.

Table 9 reports the performance overhead experienced when enabling certificates. Here we only consider instances that were solved to optimality by both versions since timeouts would bias the results in favor of the certificate. We see an increase in solving time of 56.3% on FPEASY, 22% on NUMDIFF, and of 46% on MIPLIB. This confirms the measurements presented in [13]. The increase is explained mostly by the effort to keep track of the disjunctions created during tree search and print the exact dual multipliers, and in part by an increase in dual bounding time. The reason for the latter is that bound-shift by default only provides a safe objective value. The dual multipliers needed for the certificate must be computed in a postprocessing step, which introduces the overhead in safe bounding time.

The time spent in the verification of the certificate is on average significantly lower than the time spent in the solving process. Overall, the overhead from printing and checking certificates is significant, but it does not drastically change the solvability of instances - if an instance is solvable without certificate generation then it can also be solved with certificate generation, since the solving tree remains unchanged.

Table 10 Comparison on MIPLIB 2017 benchmark set

		Original framework				New framework			
Test set	Size	Solved	Found	Time	Gap	Solved	Found	Time	Gap
All	239	18	76	6086.1	∞	53	170	3536.2	∞
Onesolved	54	18	30	3249.2	∞	53	46	304.2	∞

4.7 Overall improvements and comparison with floating-point SCIP

As a final experiment, we wanted to gauge our overall performance improvements, as well as the difference when comparing to a normal version of SCIP, i.e., running with error tolerances and computing only in floating-point arithmetic. To evaluate the overall improvements, we ran both the original framework and the revised framework with presolving and heuristics enabled on the complete MIPLIB 2017 benchmark set. The results in Table 10 show that the new framework is able to solve 35 more instances and that the mean solving time decreases by 90.6% on the subset "onesolved" of instances that could be solved to optimality by at least one solver. On more than twice as many instances at least one primal solution is found (170 vs. 76).

This raises the question how far the performance gap to a current floating-point MIP solver is, and also if we can observe the same *price of exactness* as Wolter [37] did when comparing to a reduced floating-point solver. To this end, we compared the new exact SCIP framework against the floating-point SCIP version that it is based on (SCIP 7.0.1) over the complete MIPLIB 2017 benchmark test set.

Table 11 shows the results for instances where the exact and inexact versions produced a consistent result, i.e., instances that are feasible with error tolerances but infeasible in the exact sense are discarded. The floating-point version is run once with default settings, and once in a reduced setting with all features disabled that are not yet present in the current exact SCIP version, i.e., without cutting-planes, propagation, conflict-analysis, and symmetry handling. For presolving, only the the presolver that interfaces PaPILO is run in the reduced setting.

We observe that the reduced version solved 61 instances fewer than the default, while the exact version solved 17 fewer than the reduced. That means in terms of solvability, the gap between the exact and inexact seems to be mostly due to the currently still missing solving methods.

In terms of solving time, we observe that on all instances solved to optimality by all three variants, exact SCIP is slower by a factor of 8.1 compared to default and by a factor of 3.9 compared to the reduced version. In [37], a factor of 3 was reported as the price of exactness when comparing with a reduced floating-point solver on a collection of older MIPLIB instances. This increase in price of exactness measured in our experiment may be due to testing on a more challenging set of instances. This hypothesis is supported by the fact that, on the subset of numerically difficult instances, which are also tested in [37], a factor of 9.16 was reported. The corresponding results for our framework are shown in Table 12. Here the slowdown on the alloptimal subset is a factor of 5.1 compared to the reduced version. One explanation for the smaller

Table 11 Comparison with floating-point SCIP on MIPLIB 2017 benchmark set

		SCIP default		SCIP reduced		SCIP exact	
Test set	Size	Solved	Time	Solved	Time	Solved	Time
Onesolved	137	133	213.7	62	1595.2	45	3088.4
Alloptimal	42	42	62.2	42	127.8	42	500.8

Table 12 Comparison with floating-point SCIP on NUMDIFF test set

		SCIP default		SCIP reduced		SCIP exact	
Test set	Size	Solved	Time	Solved	Time	Solved	Time
Onesolved	28	28	5.9	26	13.6	24	84.5
Alloptimal	24	24	3.8	24	7.9	24	40.3

slowdown in our revised framework may be given by the fact that presolving is able to resolve numerical difficulties on some of the instances.

5 Conclusion

We presented a substantially revised and extended solver for numerically exact mixed integer optimization that significantly improves upon the state of the art. We integrated the framework more tightly in the core of SCIP and extended it with numerically exact presolving and an exact repair heuristic.

We conducted a careful computational analysis of these changes. Overall, our changes led to a speedup factor of 10.7x compared to the original framework on the MIPLIB 2017 benchmark test set, with the strongest impact coming from the addition of symbolic presolving and switching to LP iterative refinement for exact LP solves.

We also observed, however, that the performance gap to floating-point solvers is still large. As shown by our experiments with a floating-point solver that is reduced to the same state as our current exact solver, a large portion of that performance gap can be bridged if crucial techniques such as numerically safe cutting plane separation, see, e.g., [15], are included. This must be addressed in future research.

Another potential for improvement that was identified in particular for problems with ill-conditioned LP relaxations is the extension of SoPlex by precision-boosting if LP iterative refinement on double-precision level fails breaks down. Furthermore, presolving methods need to be extended to print certificate information that can be handled by the VIPR format, such that the whole solving process can be verfified from start to finish.

With the enhancements presented in this manuscript and the easier accessibility and installation of the new framework, we are hopeful that exact rational mixed integer programming will find itself into the toolkit of a larger audience of computational mathematicians and operations researchers.

Acknowledgements We wish to thank Dan Steffy for valuable discussions on the revision of the original branch-and-bound framework, Leona Gottwald for creating PaPILO, and Antonia Chmiela for help with implementing the primal repair heuristic.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Achterberg, T.: Constraint Integer Programming. Ph.D. thesis, Technische Universität Berlin (2007)
2. Achterberg, T., Bixby, R.E., Gu, Z., Rothberg, E., Weninger, D.: Presolve reductions in mixed integer programming. Inform. J. Comput. **32**(2), 473–506 (2020). https://doi.org/10.1287/ijoc.2018.0857
3. Achterberg, T., Koch, T., Martin, A.: Branching rules revisited. Op. Res. Lett. **33**(1), 42–54 (2005). https://doi.org/10.1016/j.orl.2004.04.002
4. Achterberg, T., Wunderling, R.: Mixed integer programming: analyzing 12 years of progress. In: Jünger, M., Reinelt, G. (eds.) Facets of Combinatorial Optimization. pp. 449–481 (2013). https://doi.org/10.1007/978-3-642-38189-8_18
5. Applegate, D., Bixby, R., Chvatal, V., Cook, W.: Concorde TSP Solver (2006)
6. Applegate, D., Cook, W., Dash, S., Espinoza, D.G.: Exact solutions to linear programming problems. Op. Res. Lett. **35**(6), 693–699 (2007). https://doi.org/10.1016/j.orl.2006.12.010
7. Assarf, B., Gawrilow, E., Herr, K., Joswig, M., Lorenz, B., Paffenholz, A., Rehn, T.: Computing convex hulls and counting integer points with polymake. Math. Program. Comput. **9**(1), 1–38 (2017). https://doi.org/10.1007/s12532-016-0104-z
8. Bagnara, R., Hill, P.M., Zaffanella, E.: The parma polyhedra library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. **72**(1–2), 3–21 (2008)
9. Berthold, T.: Measuring the impact of primal heuristics. Op. Res. Lett. **41**(6), 611–614 (2013). https://doi.org/10.1016/j.orl.2013.08.007
10. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of satisfiability: volume 185 frontiers in artificial intelligence and applications. IOS Press, NLD (2009)
11. Bofill, M., Manyà, F., Vidal, A., Villaret, M.: New complexity results for Łukasiewicz logic. Soft Comput. **23**, 2187–2197 (2019). https://doi.org/10.1007/s00500-018-3365-9
12. Burton, B.A., Ozlen, M.: Computing the crosscap number of a knot using integer programming and normal surfaces. ACM Trans. Math. Softw. (2012). https://doi.org/10.1145/2382585.2382589
13. Cheung, K.K., Gleixner, A., Steffy, D.E.: Verifying Integer Programming Results. In: International Conference on Integer Programming and Combinatorial Optimization. pp. 148–160. Springer (2017). https://doi.org/10.1007/978-3-319-59250-3_13
14. Cheung, K., Gleixner, A., Steffy, D.: VIPR. Verifying Integer Programming Results. https://github.com/ambros-gleixner/VIPR (accessed May 31, 2021)
15. Cook, W., Dash, S., Fukasawa, R., Goycoolea, M.: Numerically safe gomory mixed-integer cuts. Inform. J. Comput. **21**, 641–649 (2009). https://doi.org/10.1287/ijoc.1090.0324
16. Cook, W., Koch, T., Steffy, D.E., Wolter, K.: A hybrid branch-and-bound approach for exact rational mixed-integer programming. Math. Program. Comput. **5**(3), 305–344 (2013). https://doi.org/10.1007/s12532-013-0055-6
17. Eifler, L., Gleixner, A.: Exact SCIP - a development version. https://github.com/leoneifler/exact-SCIP (accessed May 31, 2021)

18. Eifler, L., Gleixner, A.: A computational status update for exact rational mixed integer programming. In: Singh, M., Williamson, D.P. (eds.) Integer Programming and Combinatorial Optimization, pp. 163–177. Springer International Publishing, Cham (2021)
19. Eifler, L., Gleixner, A., Pulaj, J.: A safe computational framework for integer programming applied to Chvátal's conjecture (2020)
20. Espinoza, D.G.: On Linear Programming, Integer Programming and Cutting Planes. Ph.D. thesis, Georgia Institute of Technology (2006)
21. Faure, G., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: SAT Modulo the Theory of Linear Arithmetic: Exact, Inexact and Commercial Solvers. In: Kleine Büning, H., Zhao, X. (eds.) Theory and Applications of Satisfiability Testing - SAT 2008, pp. 77–90. Springer, Berlin Heidelberg, Berlin, Heidelberg (2008)
22. Gamrath, G., Anderson, D., Bestuzheva, K., Chen, W.K., Eifler, L., Gasse, M., Gemander, P., Gleixner, A., Gottwald, L., Halbig, K., Hendel, G., Hojny, C., Koch, T., Bodic, P.L., Maher, S.J., Matter, F., Miltenberger, M., Mühmer, E., Müller, B., Pfetsch, M., Schlösser, F., Serrano, F., Shinano, Y., Tawfik, C., Vigerske, S., Wegscheider, F., Weninger, D., Witzig, J.: The SCIP Optimization Suite 7.0. ZIB-Report 20-10, Zuse Institute Berlin (2020)
23. Gleixner, A., Gottwald, L., Hoen, A.: PaPILO: Parallel Presolve for Integer and Linear Optimization. https://github.com/scipopt/papilo (accessed May 28, 2021)
24. 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.D., Ozyurt, D., Ralphs, T.K., Salvagnin, D., Shinano, Y.: MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. Mathematical Programming Computation (2020), accepted for publication
25. Gleixner, A., Steffy, D.E.: Linear programming using limited-precision oracles. Math. Program. **183**, 525–554 (2020). https://doi.org/10.1007/s10107-019-01444-6
26. Gleixner, A., Steffy, D.E., Wolter, K.: Iterative refinement for linear programming. Informs. J. Comput. **28**(3), 449–464 (2016). https://doi.org/10.1287/ijoc.2016.0692
27. Granlund, T., Team, G.D.: GNU MP 6.0 Multiple Precision Arithmetic Library. Samurai Media Limited, London, GBR (2015)
28. Higham, N.J.: Accuracy and Stability of Numerical Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edn. (2002). https://doi.org/10.1137/1.9780898718027
29. Kenter, F., Skipper, D.: Integer-programming bounds on pebbling numbers of cartesian-product graphs. In: Kim, D., Uma, R.N., Zelikovsky, A. (eds.) Combinatorial Optimization and Applications. pp. 681–695 (2018). https://doi.org/10.1007/978-3-030-04651-4_46
30. Lancia, G., Pippia, E., Rinaldi, F.: Using integer programming to search for counterexamples: A case study. In: Kononov, A., Khachay, M., Kalyagin, V.A., Pardalos, P. (eds.) Mathematical Optimization Theory and Operations Research. pp. 69–84 (2020). https://doi.org/10.1007/978-3-030-49988-4
31. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
32. Neumaier, A., Shcherbina, O.: Safe bounds in linear and mixed-integer programming. Math. Program. **99**, 283–296 (2002). https://doi.org/10.1007/s10107-003-0433-3
33. Pulaj, J.: Cutting planes for families implying Frankl's conjecture. Math. Comput. **89**(322), 829–857 (2020). https://doi.org/10.1090/mcom/3461
34. Steffy, D.E., Wolter, K.: Valid linear programming bounds for exact mixed-integer programming. Inform. J. Comput. **25**(2), 271–284 (2013). https://doi.org/10.1287/ijoc.1120.0501
35. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing – SAT 2014. pp. 422–429 (2014). https://doi.org/10.1007/978-3-319-09284-3_31
36. Wilken, K., Liu, J., Heffernan, M.: Optimal instruction scheduling using integer programming. SIGPLAN Not. **35**(5), 121–133 (2000). https://doi.org/10.1145/358438.349318
37. Wolter, K.: Exact Mixed-Integer Programming. Ph.D. thesis, Technische Universität Berlin (2019)