A Computational Status Update for Exact Rational Mixed Integer Programming

Leon Eifler[1](✉) and Ambros Gleixner[1,2]

[1] Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
{eifler,gleixner}@zib.de
[2] HTW Berlin, Treskowallee 8, 10313 Berlin, Germany

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 floating-point 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 6.6x over the original framework and 2.8 times as many instances solved within a time limit of two hours.

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 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

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).

to investigate open mathematical questions include [11,12,18,28,29,32]. 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 [35].

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 [19,31,33] 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* [23], 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, by the field of satisfiability solving [34], 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 [34]. 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., [30], may also include real-valued variables, in particular for satisfiability modulo the theory of linear arithmetic. However, as pointed out also in [20], 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 [19] 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* [23]. 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* [31] 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* [33], 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 post-processed to generate a valid dual bound. The methods available for this safe bounding step are the previously described *bound-shift* [31], *project-and-shift* [33], 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.

Symbolic Presolving. The first major extension is the addition of symbolic presolving. To this end, we integrate the newly available presolving library PaPILO [25] 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 helps to compensate for the increased overhead introduced by the use of rational arithmetic. For details see [21].

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.

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.

Certainly, frequently solving this subproblem exactly can create a significant overhead compared to executing a floating-point heuristic alone, especially when a large percentage of the variables is continuous and thus cannot be fixed. Therefore, we impose working limits on the frequency of running the exact repair heuristic, which are explained in more detail in Sect. 4.

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 [24] as the strategy to solve LPs exactly, which compares favorably against incremental precision boosting [23].

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 a small amount computed from the statistically observed bounding error so far. Only when safe dual bounding fails, the objective limit 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* [27]. 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 [27, Alg. 3.2] to also account for roundoff errors $|\bar{x}^* - x^*|$ and $|\bar{a} - a|$. After doing 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.

4 Computational Study

We conduct a computational analysis to answer three 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, including the exact LP solver SoPlex, but with primal heuristics and exact presolving still disabled. In particular, we analyze the importance and performance of the different dual bounding methods.

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.

Finally, 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.

The experiments were performed on a cluster of Intel Xeon CPUs E5-2660 with 2.6 GHz and 128 GB main memory. As in [16], we use CPLEX as floating-point LP solver. Due to compatibility issues, we needed to use CPLEX 12.3.0 for the original and CPLEX 12.8.0 for the new framework. Although these versions are different, they are only used to solve floating-point LPs and have limited impact on the reported results: The vast majority of solving time is spent in the safe dual bounding methods. For exact LP solving, we use the same QSopt_ex version as in [16] and SoPlex 5.0.2. For all symbolic computations, we use the GNU Multiple Precision Library (GMP) 6.1.4 [26]. For symbolic presolving, we use PaPILO 1.0.1 [21,25]; all other SCIP presolvers are disabled.

As main test sets, 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 conduct a final comparison on the MIPLIB 2017 [22] benchmark set. 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 s for all experiments. If not stated otherwise all aggregated numbers are shifted geometric means with a shift of 0.001 s or 100 branch-and-bound nodes, respectively.

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 with SoPlex 5.0.2 as exact LP solver. 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 off within tolerances, but not with the exact the safe dual bound computed. In the new implementation we use a similar strategy, however we solve the rational LP relaxation every 5 depth levels of the tree, due to improved performance in the exact LP solver.

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 10 instances more on FPEASY and 7 more on NUMDIFF. On FPEASY, we observe a reduction of 69.8% in solving time and of 87.3% in safe dual bounding time. On NUMDIFF, we observe a reduction of 80.3% in solving time, and of 88.3% in the time spent overall in the safe dual bounding methods. We also see this significant performance improvement reflected in the two performance profiles in Fig. 1.

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	55	45	128.4	8920.1	86.8	55	38.8	5940.8	11.0
NUMDIFF	21	13	237.0	8882.7	114.6	20	46.6	6219.3	13.4

Table 2. Comparison of safe dual bounding techniques

		Original framework			New framework		
Test set	Stats	bshift	pshift	exlp	bshift	pshift	exlp
FPEASY	Calls/node	0.92	0.44	0.28	0.53	0.39	0.06
	Time/call [s]	0.0026	0.0022	0.050	0.0072	0.0066	0.010
	Time/solving time	2.9%	40.3%	32.1%	10.8%	27.8%	4.4%
NUMDIFF	Calls/node	0.78	0.36	0.52	0.36	0.39	0.28
	Time/call [s]	0.0055	0.0036	0.4197	0.0247	0.0356	0.1556
	Time/solving time	1.4%	22.7%	62.2%	5.2%	24.8%	40.1%

We identify a more aggressive use of project-and-shift and faster exact LP solves as the two key factors for this improvement. In the original framework, project-and-shift is restricted to instances that had less than 10000 nonzeros. One 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. With the improvements in exact LP performance, it proved beneficial to remove this working limit in the new framework.

The effect of this change can also be seen in the detailed analysis of bounding times given in Table 2. 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 over all instances where the respective bounding method was called at least once.

The fact that time per call for project-and-shift (“pshift”) in the new framework increased by a factor of 3 (FPEASY) and 9.9 (NUMDIFF) is for the reason discussed above—it is now also called on larger instances. This is beneficial overall since it replaces many slower exact LP calls. The decrease in exact LP solving time per call (“exlp”) by a factor of 2.7 (NUMDIFF) and 5 (FPEASY) can also partly be explained by this change, and partly by an overall performance improvement in exact LP solving due to the use of LP iterative refinement [24]. The increase in bound-shift time (“bshift”) 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. Finally, we observe a decrease in the total number of safe bounding calls per node. One reason is that we now disable bound-shift dynamically if its success rate drops below 20%.

Overall, we see a notable speedup and more solved instances, mainly due to the better management of dual bounding methods and faster exact LP solving.

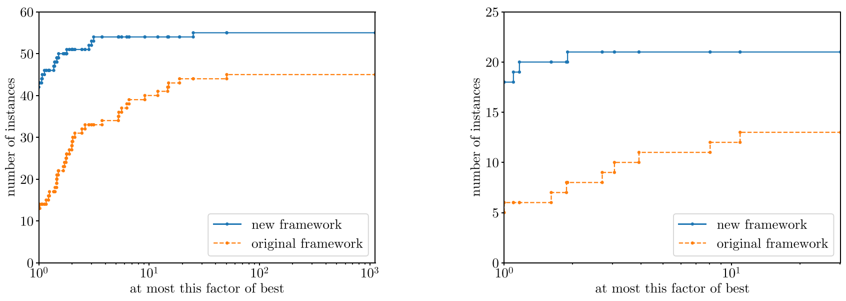

Fig. 1. Performance profiles comparing solving time of original and new framework without presolving and heuristics for FPEASY (left) and NUMDIFF (right)

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 less 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 3 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 the 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 0.86% (FPEASY) and 2.1% (NUMDIFF) of the total solving time, as seen in Table 4. Exploiting parallelism in presolving provided no measureable benefit for floating-point presolving, but reduced symbolic presolving time by 44% (FPEASY) to 43.8% (NUMDIFF). However, this benefit can be more pronounced on individual instances, e.g., on `nw04`, where parallelization reduces the time for rational presolving by a factor of 6.4 from 1770 to 277 s.

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 4. Enabling presolving solves 3 more instances on FPEASY and 20 more instances on NUMDIFF. We observe a reduction in solving time of 39.4% (FPEASY) and 72.9% (NUMDIFF). The stronger impact on NUMDIFF is correlated with the larger number of reductions observed in Table 3.

Table 3. 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	8.5	3.5	10.4	0.25	3.2	8.5	3.5	10.4
	20	0.01	3.2	8.5	3.5	10.4	0.14	3.2	8.5	3.5	10.4
NUMDIFF	1	0.04	8.3	53.8	55.7	51.4	0.89	7.2	41.4	42.9	55.8
	20	0.04	8.3	53.8	55.7	51.4	0.50	7.2	41.4	42.9	55.8

Table 4. 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	165	42.1	6145.3	168	25.5 (0.22)	4724.1
NUMDIFF	91	66	216.6	7237.2	86	58.7 (1.23)	2867.2

Primal Heuristics. To improve primal performance, we enabled all SCIP heuristics that the floating-point version executes by default. To limit the fraction of solving time for the repair heuristic described in Sect. 3, the repair heuristic is only allowed to run at any point in the solve, if it was called at most half as often as the exact LP calls for safe dual bounding. Furthermore, 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. Whenever the repair step is not executed, the floating-point solutions are checked directly for exact feasibility.

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 5. The repair heuristic is effective at finding feasible solutions with a success rate of 46.9% (FPEASY) and 25.6% (NUMDIFF). 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, and performance on these instances would quickly deteriorate if the working limits were relaxed.

Table 5. Statistics of repair heuristic for instances where repair step was called

		Time				
Test set	Size	Total solving	Repair	Fail	Success	Success rate
FPEASY	82	39.8	0.0020	0.0017	0.0003	46.9%
NUMDIFF	42	383.6	0.0187	0.0077	0.0062	25.6%

Table 6. 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	82	32.5	0.75	2351.8	32.6	0.10	2037.2
NUMDIFF	41	101.7	4.77	8670.7	103.1	1.30	9093.6

Table 6 shows the overall performance impact of enabling heuristics over all instances that could be solved by at least one setting. On both sets, we see almost no change in total solving time. On FPEASY, the time to find the first solution decreases by 86.7% and the primal integral decreases by 13.4%. The picture is slightly different on the numerically difficult test set. Here, the time to find the first solution decreases by 72.7%, while the primal integral increases by 4.9%.

The worse performance and success rate on NUMDIFF is expected, considering that this test set was curated to contain instances with numerical challenges. On those instances floating-point heuristics find solutions that might either not be feasible in exact arithmetic or are not possible to fix for the repair heuristic. In both test sets, the repair heuristic was able to find solutions, while not imposing any significant overhead in solving time.

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.

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 fails. For example, on instance x_4 from NUMDIFF, the original framework claimed infeasibility at the root node, and while the instance is indeed infeasible, we found the reasoning for this to be incorrect due to the use of a certificate.

Table 7 reports the performance overhead 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 101.2% on FPEASY and of 51.4% on NUMDIFF. This confirms the measurements presented in [13]. The increase is explained in part by the effort to keep track of the tree structure 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. This overhead is larger on FPEASY, since bound-shift is called more often. 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.

Table 7. 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	53	32.6	9.1	65.6	16.5	0.9	103.9%
NUMDIFF	21	41.6	11.9	63.0	18.0	0.5	52.6%

Table 8. Comparison on MIPLIB 2017 benchmark set

		Original framework				New framework			
Test set	Size	Solved	Found	Time	Gap	Solved	Found	Time	Gap
All	240	17	74	6003.6	∞	47	167	3928.1	∞
Both	66	16	66	4180.0	67.9%	29	66	1896.2	33.8%
Onesolved	49	17	31	3317.5	∞	47	47	505.1	∞

Performance Comparison on MIBLIB 2017. As a final experiment, we wanted to evaluate the performance on a more ambitious and diverse test set. To that end, we ran both the original framework and the revised framework with presolving and heuristics enabled on the recent MIPLIB 2017 benchmark set. The results in Table 8 show that the new framework solved 30 instances more and the mean solving time decreased by 84.8% 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 was found (167 vs. 74). On the subset of 66 instances that had a finite gap for both versions, the new algorithm achieved a gap of 33.8% in arithmetic mean compared to 67.9% in the original framework.

To conclude, we presented a substantially revised and extended solver for numerically exact mixed integer optimization that significantly improves upon the existing state of the art. We also observe, however, that the performance gap to floating-point solvers is still large. This is not surprising, given that crucial techniques such as numerically safe cutting plane separation, see, e.g., [15], are not yet included. This must be addressed in future research.

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.

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. INFORMS 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. Oper. 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. Springer, Heidelberg (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. Oper. Res. Lett. **35**(6), 693–699 (2007). https://doi.org/10.1016/j.orl.2006.12.010
7. Assarf, B., et al.: 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. Oper. 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, Amsterdam (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. **39**(1) (2012). https://doi.org/10.1145/2382585.2382589
13. Cheung, K.K.H., Gleixner, A., Steffy, D.E.: Verifying integer programming results. In: Eisenbrand, F., Koenemann, J. (eds.) IPCO 2017. LNCS, vol. 10328, pp. 148–160. Springer, Cham (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 11 Nov 2020
15. Cook, W., Dash, S., Fukasawa, R., Goycoolea, M.: Numerically safe gomory mixed-integer cuts. INFORMS 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 11 Nov 2020
18. Eifler, L., Gleixner, A., Pulaj, J.: A safe computational framework for integer programming applied to Chvátal's conjecture (2020)
19. Espinoza, D.G.: On linear programming, integer programming and cutting planes. Ph.D. thesis, Georgia Institute of Technology (2006)
20. 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.) SAT 2008. LNCS, vol. 4996, pp. 77–90. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79719-7_8
21. Gamrath, G., et al.: The SCIP Optimization Suite 7.0. ZIB-Report 20–10, Zuse Institute Berlin (2020)
22. Gleixner, A., et al.: MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library. Math. Program. Comput. 1–48 (2021). https://doi.org/10.1007/s12532-020-00194-3
23. 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
24. 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
25. Gottwald, L.: PaPILO – Parallel Presolve for Integer and Linear Optimization. https://github.com/lgottwald/PaPILO. Accessed 9 Sep 2020
26. Granlund, T., Team, G.D.: GNU MP 6.0 Multiple Precision Arithmetic Library. Samurai Media Limited, London, GBR (2015)
27. Higham, N.J.: Accuracy and Stability of Numerical Algorithms, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia (2002). https://doi.org/10.1137/1.9780898718027

28. Kenter, F., Skipper, D.: Integer-programming bounds on pebbling numbers of Cartesian-product graphs. In: Kim, D., Uma, R.N., Zelikovsky, A. (eds.) COCOA 2018. LNCS, vol. 11346, pp. 681–695. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-04651-4_46
29. 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.) MOTOR 2020. LNCS, vol. 12095, pp. 69–84. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-49988-4_5
30. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
31. 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
32. Pulaj, J.: Cutting planes for families implying Frankl's conjecture. Math. Comput. **89**(322), 829–857 (2020). https://doi.org/10.1090/mcom/3461
33. Steffy, D.E., Wolter, K.: Valid linear programming bounds for exact mixed-integer programming. INFORMS J. Comput. **25**(2), 271–284 (2013). https://doi.org/10.1287/ijoc.1120.0501
34. 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.) SAT 2014. LNCS, vol. 8561, pp. 422–429. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_31
35. 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