The SCIP Optimization Suite 10.0

Christopher Hojny · Mathieu Besançon · Ksenia Bestuzheva
Sander Borst · Antonia Chmiela, João Dionísio · Johannes Ehls
Leon Eifler · Mohammed Ghannam · Ambros Gleixner
Adrian Göß · Alexander Hoen · Jacob von Holly-Ponientzietz
Rolf van der Hulst · Dominik Kamp · Thorsten Koch
Kevin Kofler · Jurgen Lentz · Marco Lübbecke
Stephen J. Maher · Paul Matti Meinhold · Gioni Mexi
Til Mohr · Erik Mühmer · Krunal Kishor Patel
Marc E. Pfetsch · Sebastian Pokutta · Chantal Reinartz Groba
Felipe Serrano · Yuji Shinano · Mark Turner · Stefan Vigerske
Matthias Walter · Dieter Weninger · Liding Xu [*]

November 23, 2025

Abstract The SCIP Optimization Suite provides a collection of software packages for mathematical optimization, centered around the constraint integer programming (CIP) framework SCIP. This report discusses the enhancements and extensions included in SCIP Optimization Suite 10.0. The updates in SCIP 10.0 include a new solving mode for exactly solving rational mixed-integer linear programs, a new presolver for detecting implied integral variables, a novel cut-based conflict analysis and separator for flower inequalities, two new heuristics, a novel tool for explaining infeasibility, a new interface for nonlinear solvers as well as improvements in symmetry handling, branching strategies, and SCIP's Benders' decomposition framework. SCIP Optimization Suite 10.0 also includes new and improved features in the the presolving library PaPILO, the parallel framework UG, and the decomposition framework GCG. Moreover, the SCIP Optimization Suite 10.0 contains MIP-DD, the first open-source delta debugger for mixed-integer programming solvers. These additions and enhancements have resulted in an overall performance improvement of SCIP in terms of solving time, number of nodes in the branch-and-bound tree, as well as the reliability of the solver.

Keywords Constraint integer programming · linear programming · mixed-integer linear programming · mixed-integer nonlinear programming · optimization solver · branch-and-cut · branch-and-price · column generation · parallelization · mixed-integer semidefinite programming

Mathematics Subject Classification 90C05 · 90C10 · 90C11 · 90C30 · 90C90 · 65Y05

[*]Extended author information is available at the end of the paper. The work for this article was partially supported through the *Research Campus MODAL* funded by the German Federal Ministry of Research, Technology, and Space (fund numbers *05M14ZAM*, *05M20ZBM*) and the *Deutsche Forschungsgemeinschaft (DFG)* through the *DFG Cluster of Excellence MATH+*. It has also been partly supported the project *Making Mixed-Integer Programming Solvers Smarter and Faster using Network Matrices* (with project number *OCENW.M20.151*) of the research programme *NWO Open Competition Domain Science – M* which is (partly) financed by the Dutch Research Council (NWO).

1 Introduction

The SCIP Optimization Suite comprises a set of complementary software packages designed to model and solve a large variety of mathematical optimization problems:

– the constraint integer programming solver SCIP [3], a solver for mixed-integer linear and nonlinear programs as well as a flexible framework for branch-cut-and-price,

– the simplex-based linear programming solver SoPlex [110],

– the modeling language Zimpl [70],

– the presolving library PaPILO for linear and mixed-integer linear programs,

– the automatic decomposition solver GCG [42], and

– the UG framework for parallelization of branch-and-bound solvers [100].

All six tools are freely available as open-source software packages; SCIP 10.0, SoPlex 8.0, PaPILO 3.0, and GCG 4.0 are licensed under the Apache 2.0 license, whereas Zimpl 3.7.0 and UG 1.0 make use of the GNU Lesser General Public License. A notable extension to the SCIP Optimization Suite is the mixed-integer semidefinite programming solver SCIP-SDP [41]. Moreover, the delta debugging tool MIP-DD [55] can be used to assist the development and debugging of optimization software like SCIP.

New Developments and Structure of the Paper The goal of this report is to highlight the features becoming available in the latest release of the SCIP Optimization Suite, where the focus is on the developments of SCIP. After providing a short background on the problems that can be handled by SCIP, Section 2 investigates the change of performance between SCIP 9.0 and SCIP 10.0 for mixed-integer linear and nonlinear programs. Briefly summarized, SCIP 10.0 is more performant than SCIP 9.0 both in terms of running time and number of solved instances, where the performance improvement is more pronounced for nonlinear problems. Section 3 forms the main part of this report and highlights the new features and technical improvements becoming available in SCIP 10, namely

– a new solving mode that allows to solve rational MILPs exactly, i.e., without numerical tolerances and without being affected by floating-point roundoff errors (Section 3.1);

– a new presolver for detecting implied integral variables and a more flexible specification of implied integrality information (Section 3.2);

– generalizations of symmetry handling methods to reflection symmetries (Section 3.3);

– cut-based conflict analysis (Section 3.4);

– a new separator plugin for flower inequalities derived from multilinear problems (Section 3.5);

– two new primal heuristics that exploit decompositions provided by users (Section 3.6);

– improved branching strategies (Section 3.7);

– enhancing SCIP's Benders' decomposition framework by a more flexible way to formulate problems and an improved detection of master linking variables (Section 3.8);

– a tool to find explanations for infeasibility via so-called irreducible infeasible subsystems (Section 3.9);

– an interface to the nonlinear programming solver CONOPT (Section 3.10);

– two technical improvements: a new constraint handler to avoid infeasibilities due to aggregations from presolving and a JSON export of SCIP's statistics (Section 3.11).

Section 4 is devoted to improvements of GCG, including new data structures and file formats for storing decompositions, newly developed solvers for pricing problems, parallelization of pricing, and support for handling branching and cutting decisions formulated directly in the extended formulation variables. Section 5 gives a short update on SCIP-SDP. Section 6 highlights the latest developments of PaPILO improving performance and memory management. Section 7 gives a very brief summary of the capabilities of SoPlex, and Sections 8 and 9 briefly summarize the latest changes in UG and Zimpl, respectively. In Section 10, new features of SCIP's various interfaces are discussed, and Section 11 concerns the delta debugger MIP-DD. Section 12 discusses the newly developed PBSolver, a SCIP-based solver that is tailored for Pseudo-Boolean problems and won several categories of the Pseudo-Boolean Competition 24.

Background SCIP is designed as a solver for *constraint integer programs* (CIPs), a generalization of mixed-integer linear and nonlinear programs (MILPs and MINLPs). CIPs are finite-dimensional optimization problems with arbitrary constraints and a linear objective function that satisfy the following property: if all integer variables are fixed, the remaining subproblem must form a linear or nonlinear program (LP or NLP). To solve CIPs, SCIP constructs relaxations—typically linear relaxations, but also nonlinear relaxations are possible, or relaxations based on semidefinite programming for SCIP-SDP. If the relaxation solution is not feasible for the current subproblem, an *enforcement* procedure is called that resolves the infeasibility, for example by branching or by separating cutting planes.

The most important subclass of CIPs that are solvable with SCIP are *mixed-integer programs* (MIPs) which can be purely linear (MILPs) or contain nonlinear terms in the constraints or the objective function (MINLPs). MILPs are optimization problems of the form

$$\begin{aligned} \min \quad & c^\top x && \\ \text{s.t.} \quad & Ax \geq b, && \\ & \ell_i \leq x_i \leq u_i && \text{for all } i \in \mathcal{N}, \\ & x_i \in \mathbb{Z} && \text{for all } i \in \mathcal{I}, \end{aligned} \tag{1}$$

defined by $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, ℓ, $u \in \overline{\mathbb{R}}^n$, and the index set of integer variables $\mathcal{I} \subseteq \mathcal{N} := \{1, \ldots, n\}$. The usage of $\overline{\mathbb{R}} := \mathbb{R} \cup \{-\infty, \infty\}$ allows for variables that are free or bounded only in one direction (we assume that variables are not fixed to $\pm\infty$). In contrast, MINLPs are optimization problems of the form

$$\begin{aligned} \min \quad & f(x) && \\ \text{s.t.} \quad & g_k(x) \leq 0 && \text{for all } k \in \mathcal{M}, \\ & \ell_i \leq x_i \leq u_i && \text{for all } i \in \mathcal{N}, \\ & x_i \in \mathbb{Z} && \text{for all } i \in \mathcal{I}, \end{aligned} \tag{2}$$

where the functions $f \colon \mathbb{R}^n \to \mathbb{R}$ and $g_k \colon \mathbb{R}^n \to \mathbb{R}$, $k \in \mathcal{M} := \{1, \ldots, m\}$, are possibly nonconvex. Within SCIP, it is assumed that f is linear and that g_k, $k \in \mathcal{M}$, is specified explicitly in algebraic form using a known set of base expressions. Due to its design as a solver for CIPs, SCIP can be extended by plugins for more general or problem-specific classes of optimization problems. The core of SCIP is formed by a central branch-cut-and-price algorithm that utilizes an LP as the default relaxation which can be solved by a number of different LP solvers, controlled through a uniform *LP interface*. To be able to handle any type of constraint, a *constraint handler* interface is provided. This interface allows to integrate new constraint types, and provides support for many different well-known types of constraints out of the box. Further solving methods like primal heuristics, branching rules, and cutting plane separators can also be integrated as plugins with a pre-defined interface. SCIP comes with many such plugins needed to achieve a good MILP and MINLP performance. In addition to plugins supplied as part of the SCIP distribution, new plugins can be created by users. The design approach and solving process is described in detail by Achterberg [1].

Although SCIP is a standalone solver, it interacts closely with the other components of the SCIP Optimization Suite. Zimpl is integrated into SCIP as a reader plugin, making it possible to read Zimpl problem instances directly by SCIP. PaPILO is integrated into SCIP as an additional presolver plugin. The LPs that need to be solved as relaxations in the branch-and-bound process are by default solved with SoPlex, and it is possible to replace SoPlex by other LP solvers. Interfaces to most actively developed external LP solvers exist, and new interfaces can be added by users. GCG extends SCIP to automatically detect problem structure and generically apply decomposition algorithms based on the Dantzig-Wolfe or the Benders' decomposition schemes. Finally, the default instantiations of the UG framework use SCIP as a base solver in order to perform branch-and-bound in parallel computing environments with shared or distributed memory architectures.

2 Overall Performance Improvements for MILP and MINLP with SCIP

A major use of the SCIP Optimization Suite is as an out-of-the-box solver for mixed-integer linear and nonlinear programs. Therefore, the performance of SCIP on MILP and MINLP instances is of particular interest during the development process. In this section, we present computational experiments to assess the performance of SCIP 10.0 in comparison with the previous major release, SCIP 9.0, and the latest bugfix release, SCIP 9.2.4. The reason for comparing SCIP 10.0 with these two previous versions is to highlight the effect of both increasing robustness of SCIP by fixing bugs (comparison with SCIP 9.2.4) and new features (comparison with SCIP 9.0).

2.1 Experimental Setup

We compare SCIP 10.0, including SoPlex 8.0.0 and PaPILO 3.0.0, with SCIP 9.0 (including SoPlex 7.0.0 and PaPILO 2.2.0) as well as SCIP 9.2.4 (including SoPlex 7.1.6 and PaPILO 2.4.4). All versions of SCIP were built with GCC 10.2.1. Further software packages linked to SCIP include the NLP solver Ipopt 3.14.18 built with HSL MA27 as linear system solver, Intel MKL as linear algebra package, and the algorithmic differentiation code CppAD 20180000.0. For symmetry detection, SCIP 10.0 uses the external software nauty 2.8.8 combined with the symmetry preprocessor sassy 2.0; SCIP 9.2.4 uses nauty 2.8.8 with sassy 1.1, and SCIP 9.0 uses bliss 0.77 with sassy 1.1. The time limit was set to 5400 s for MILP and to 3600 s for the MINLP runs. Note that for MINLP, an instance is considered solved when either a relative primal-dual gap of 10^{-4}, or an absolute gap of 10^{-6} is reached.

The MILP instances are selected from the MIPLIB 2003, 2010, and 2017 [46] sets, as well as the COR@L instance set, and include all instances that could be solved by previous releases, for a total of 349 instances. The MINLP instances are selected similarly from the MINLPLib[1] library, for a total of 169 instances. All performance runs are carried out on identical machines with Intel Xeon Gold 5122 @ 3.60 GHz and 96 GB RAM. A single run is carried out on each machine in single-threaded mode. Each MILP instance is solved with SCIP using three different seeds for random number generators, while each MINLP instance is solved using five different seeds. This results in a testset of 1047 MILPs and 845 MINLPs.

[1]https://www.minlplib.org

The indicators of interest for our comparison on a given subset of instances are the number of solved instances, the shifted geometric mean of the number of branch-and-bound nodes, and the shifted geometric mean of the solving time. The *shifted geometric mean* of values $t_1, \dots, t_n$ is

$$\left(\prod_{i=1}^{n}(t_i + s)\right)^{1/n} - s.$$

The shift s is set to 100 nodes and 1 second, respectively.

In Tables 1 and 3, we present the results of the computational experiments for MILP and MINLP comparing SCIP 10.0 and SCIP 9.0; Tables 2 and 4 show the results of the comparison of SCIP 10.0 and SCIP 9.2.4. We present these statistics for several subsets of instances. The subset allcontains all instances of the testset excluding those with numerically inconsistent results. The subset “affected” contains all instances where solvers differ in the number of dual simplex iterations. The brackets $[t, T]$ collect the subsets of instances which were solved by at least one solver and for which the maximum solving time (among both solver versions) is at least t seconds and at most T seconds, where T is usually equal to the time limit. With increasing t, this provides a hierarchy of subsets of increasing difficulty. The subset “both-solved” contains instances that can be solved by both versions within the time limit. Columns “instances” and “solved” refer to the number of instances in a subset of instances and the number of solved instances, respectively. Moreover, columns “time” and “nodes” provide the shifted geometric mean of the running time and number of nodes, respectively, needed by the different versions.

2.2 MILP Performance

Table 1 summarizes the results comparing the performance of SCIP 10.0 against SCIP 9.0 on the MILP testset. Using SCIP 10.0 instead of SCIP 9.0 slightly increases the number of solved instances from 886 to 888 and decreases the mean running time by approximately 2 %. Despite the reduced running time, the number of nodes in SCIP’s branch-and-bound tree increases by 3 %. Since the mean number of nodes is relatively small though, the increased size of the branch-and-bound tree is only moderate in absolute numbers.

On the various subsets of instances, one can observe that SCIP’s performance improvement is better for the harder instances that require at least ten seconds to be solved by both versions. Among all test sets, including both-solved, SCIP 10.0 is at least 3 % faster than SCIP 9.0; for the [100,timelim] subset even an 8 % performance improvement can be observed and also the number of nodes reduces by 3 %. That is, SCIP’s performance gain on the harder instances mainly contributes to its overall performance improvement in comparison with SCIP 9.0.

Comparing SCIP 10.0 with SCIP 9.2.4 (Table 2), similar observations as before can be made, however, with more pronounced performance improvements. The overall performance of SCIP 10.0 is 4 % better than the performance of SCIP 9.2.4, where again the biggest performance improvements are achieved for the harder instances. For example, for the [100,timelim] and [1000,timelim] subsets, SCIP 10.0 has a 9 % and 11 %, respectively, lower running time than SCIP 9.2.4. This also results in a reduced number of nodes in the branch-and-bound trees on most subsets.

To summarize, SCIP 10.0 has become more reliable than SCIP 9.0 based on the bugfixes that have been introduced in SCIP 9.2.4. While some of these bugfixes caused a slowdown of SCIP 9.2.4 in comparison with SCIP 9.0, the additional features of SCIP 10.0 yield an overall performance improvement both in comparison with SCIP 9.0 and SCIP 9.2.4.

Table 1: Performance of SCIP 10.0 and SCIP 9.0 for MILP instances.

		SCIP 10.0.0+SoPlex 8.0.0			SCIP 9.0.0+SoPlex 7.0.0			relative	
Subset	instances	solved	time	nodes	solved	time	nodes	time	nodes
all	1046	888	232.6	2659	886	236.3	2614	0.98	1.02
affected	827	796	195.6	2823	794	198.8	2733	0.98	1.03
[1,timelim]	890	859	174.7	2293	857	177.9	2224	0.98	1.03
[10,timelim]	815	784	242.1	2912	782	248.6	2825	0.97	1.03
[100,timelim]	573	542	569.1	6337	540	617.3	6528	0.92	0.97
[1000,timelim]	228	197	1919.5	19663	195	1981.4	19049	0.97	1.03
both-solved	855	855	119.3	1695	855	122.4	1671	0.97	1.01

Table 2: Performance of SCIP 10.0 and SCIP 9.2.4 for MILP instances.

		SCIP 10.0.0+SoPlex 8.0.0			SCIP 9.2.4+SoPlex 7.1.6			relative	
Subset	instances	solved	time	nodes	solved	time	nodes	time	nodes
all	1046	888	232.6	2659	883	241.6	2669	0.96	1.00
affected	783	761	202.7	3015	756	212.3	3043	0.96	0.99
[1,timelim]	881	859	168.7	2225	854	176.4	2243	0.96	0.99
[10,timelim]	805	783	234.9	2844	778	247.3	2862	0.95	0.99
[100,timelim]	562	540	561.0	6072	535	616.8	6413	0.91	0.95
[1000,timelim]	221	199	1771.0	17321	194	2000.8	19337	0.89	0.90
both-solved	861	861	123.0	1770	861	126.4	1759	0.97	1.01

2.3 MINLP Performance

Table 3 provides an overview of the results for SCIP 10.0 and SCIP 9.0 on the MINLP testset. As for the MILP testset, SCIP 10.0 slightly increases the number of solved instances from 800 to 802. For all tested subsets of instances, the number of nodes in the branch-and-bound tree decreases by at least 2 %. On the entire testset, SCIP 10.0 is 6 % faster than SCIP 9.0; on the subset of affected instances, the performance improvement is even 8 %.

Distinguishing the running time improvements based on the difficulty of instances, the highest performance gain is achieved for instances that require at least 1000 s, where it materializes in a speed-up of 20 %. This high performance gain can be explained by SCIP 10.0 solving two more instances than SCIP 9.0, i.e., every instance that was solvable by SCIP 9.0 can also be solved by SCIP 10.0 within the time limit. Restricting to the subset both-optimal, Table 3 shows that SCIP 10.0 is consistently faster on these instances, achieving a performance improvement of 5 %. That is, SCIP 10.0 both improves the running time and allows to solve more instances.

Comparing SCIP 10.0 with SCIP 9.2.4 on the MINLP testset (Table 4), we observe a similar trend as for the MILP testset: both the performance improvements and reduction of number nodes in the branch-and-bound trees are more pronounced. On the entire testset, SCIP 10.0 achieves a performance improvement of 9 %, and on the [1000,timelim] subset the performance even improves by 22 %; the number of nodes decreases by at least 6 %. The additional features of SCIP 10.0 and bugfixes introduced by SCIP 9.2.4 thus also yield for the MINLP testset a more reliable and faster code base.

3 SCIP

3.1 A Numerically Exact Solving Mode for MILPs

As of today, virtually all available MIP solvers rely on floating-point arithmetic coupled with numerical error tolerances to achieve a tradeoff of numerical accuracy and performance. While typically a lot of development effort is invested into ensuring high

Table 3: Performance of SCIP 10.0 and SCIP 9.0 for MINLP instances

		SCIP 10.0.0+CPLEX 12.10.0.0			SCIP 9.0.0+CPLEX 12.10.0.0			relative	
Subset	instances	solved	time	nodes	solved	time	nodes	time	nodes
all	809	802	17.9	2375	800	19.0	2433	0.94	0.98
affected	696	696	20.1	2515	694	21.7	2588	0.92	0.97
[1,timelim]	720	720	22.9	3164	718	24.5	3254	0.93	0.97
[10,timelim]	444	444	66.7	7341	442	74.5	7723	0.89	0.95
[100,timelim]	168	168	366.6	44746	166	413.5	48104	0.89	0.93
[1000,timelim]	52	52	1094.0	130587	50	1362.3	134476	0.80	0.97
both-solved	800	800	17.0	2233	800	17.9	2274	0.95	0.98

Table 4: Performance of SCIP 10.0 and SCIP 9.2.4 for MINLP instances.

		SCIP 10.0.0+CPLEX 12.10.0.0			SCIP 9.2.4+CPLEX 12.10.0.0			relative	
Subset	instances	solved	time	nodes	solved	time	nodes	time	nodes
all	816	804	18.5	2399	804	20.4	2547	0.91	0.94
affected	569	569	21.6	2255	569	24.5	2460	0.88	0.92
[1,timelim]	721	721	22.9	3172	721	25.5	3384	0.90	0.94
[10,timelim]	445	445	66.7	7308	445	77.9	8080	0.86	0.90
[100,timelim]	175	175	330.4	38810	175	407.6	46111	0.81	0.84
[1000,timelim]	52	52	1156.7	145999	52	1480.6	161852	0.78	0.90
both-solved	804	804	17.0	2246	804	18.8	2389	0.91	0.94

numerical robustness, by design floating-point solvers cannot guarantee that their results are unaffected by accumulated roundoff errors, nor can they produce independently verifiable certificates of correctness.

In SCIP 10.0 users now for the first time have the option to solve MILPs with rational input data in a numerically exact solving mode subject to no numerical tolerances. The exact solving mode relies on the software packages GMP, MPFR, and Boost. If these dependencies are satisfied while building SCIP, the exact solving mode can be activated at runtime simply by setting the parameter `exact/enabled` to `true` before the problem instance is loaded. In addition, SCIP 10.0 can log a proof of correctness of the LP-based branch-and-bound process by setting the parameter `certificate/filename`. SCIP then produces a certificate file in `VIPR` [22] format that encodes SCIP's reasoning for the claimed primal and dual bounds. The correctness of this certificate can then be verified by independent proof checkers.

As a guiding principle, the exact solving mode follows a hybrid-precision approach and replaces expensive symbolic computations at many places by numerically safe floating-point computations based on directed rounding. In SCIP 10.0, the exact solving mode is restricted to mixed-integer linear programs and supports the following features:

– rational presolving using PaPILO [36, 47],

– safe dual bounding and reliability pseudocost branching [25, 61],

– safe separation of Gomory mixed-integer cuts [37],

– safe dual proof analysis and constraint propagation [18],

– exact post-processing of solutions from all floating-point primal heuristics [36],

– certification of the LP-based branch-and-bound process [22] except presolving, which can be certified independently for binary programs with integer data [54].

For a detailed description of the theory and performance of the various exact solving algorithms, we refer to the above references.

The following sections give an overview of the parts of SCIP added or modified for implementing the exact solving mode, as well as the general ideas behind the new features. In order to ensure that addition of an unsafe user plugin does not compromise the correctness of the exact solving mode, all critical plugin types have been equipped with a flag that stores whether a plugin of that type can be safely used in exact solving mode. As a general design principle, the SCIP core executes callbacks only of those plugins that are explicitly marked as safe to use in exact solving mode, which is typically done in the plugin's inclusion method.

3.1.1 Exact Instance Readers and a Wrapper for Rational Numbers

The first step in the solving process that can result in numerical errors is the reading of the problem instance. SCIP 10.0 provides extensions to the readers for MPS, LP, CIP, OPB/WBO, and ZIMPL files to read the problem instance in exact arithmetic. This means that those types of problems can contain any combination of floating-point, rational, and integer coefficients, all of which will be read exactly. Note that the exact solving mode needs to be enabled prior to the creation of the problem. The exact readers will then parse all values of bounds and coefficients as rational numbers and add exact variables and constraints to SCIP, see Section 3.1.2. For storing and computing with rational numbers, SCIP 10.0 comes with a new data structure `SCIP_RATIONAL`, which acts as a wrapper to combine functionality of the Boost multiprecision library [17], GMP [48], and MPFR [40, 85].

3.1.2 Exact Variable Data and a New Handler for Exact Linear Constraints

In SCIP 10.0, the variable structure contains a new substructure, allocated only in exact solving mode, that keeps rational versions of the objective coefficient, the global and local domains, and data for (multi-)aggregations, the exact LP, and the certificate. As before, each variable is initially created with floating-point data. If the floating-point bounds and objective coefficient are correct, the exact data can be initialized with the floating-point data; otherwise, the floating-point data is overwritten. Note that the floating-point bounds are always maintained as a safe outward rounding of the exact bounds so they can be used algorithmically during constraint propagation and dual bounding.

Following SCIP's constraint-based view, also a new constraint handler has been added for exact linear constraints. In the exact solving mode, this constraint handler replaces the standard linear constraint handler, and ensures that there is no loss of accuracy in any of the callbacks. The new handler for exact linear constraints manages the exact representation of linear constraints, their addition to LP data structures, exact feasibility checking and enforcement of solutions, constraint propagation routines, as well as some numerically safe techniques to improve efficiency such as a running error analysis for faster feasibility checks [36].

3.1.3 Exact Presolving using PaPILO

Because the presolving library PaPILO [47] is fully templatized with respect to the number type of its input data and all arithmetical operations, it is ideally suited for performing presolving in SCIP's exact solving mode. For SCIP 10.0, the presolver `presol_milp`, which interfaces to PaPILO, has been extended to pass a rational representation of the problem instance to PaPILO, set PaPILO's numerical tolerances to zero, and call PaPILO to perform all presolving steps in exact rational arithmetic. Although presolving in exact rational arithmetic is slower than in floating-point arithmetic, the additional overhead can be reduced by executing PaPILO in parallel, which can be enabled via parameter `presolving/milp/nthreads`, see [36] for a detailed evaluation.

3.1.4 LP Relaxation, Exact LP Solving, and Numerically Safe Dual Bounds

While SCIP's standard LP structure is unmodified and maintains a floating-point approximation of the exact data, SCIP 10.0 comes with a new exact LP structure that manages a rational version of the LP relaxation. Similar to the LP interface for floating-point arithmetic, an exact LP solver interface is provided that can currently be used with the SoPlex and QSopt_ex [39] LP solvers. Both support solving LPs exactly, with SoPlex being the default exact LP solver in SCIP 10.0.

Because solving an LP exactly for every LP relaxation would be computationally expensive, exact SCIP employs a selection of numerically safe dual bounding techniques to avoid calling the exact LP solver for every LP relaxation. These methods are called *bound shift* and *project and shift*, see [25, 61] for detailed descriptions of these methods. The exact LP solver is called as a fallback if one of these cheaper methods fails, which may happen, e.g., if bound shift encounters unbounded variables. By default, exact SCIP also calls the exact LP solver in one of the following situations in order to improve overall performance:

- at the end of the root node,
- at depth levels 2^k for $k = 1, 2, 3, \ldots$,
- when determining unboundedness,
- when the floating-point LP solution is close to being integer feasible,
- when the floating-point LP bound is close to the cutoff bound, or
- when all integer variables are fixed.

3.1.5 Reliability Pseudocost Branching and Numerically Safe Cutting Planes

In exact solving mode, if the LP relaxation solution violates integrality by more than the floating-point feasibility tolerance, then the `integral` constraint handler calls the branching rule with highest priority that is marked exact. In SCIP 10.0, the only branching rule made exact is reliability pseudocost branching, which is also the default branching rule in floating-point mode. We refer to [25] for a detailed description and computational study. Note that the current version calls the floating-point LP solver for strong branching and the resulting bounds are not made safe. This means the results are currently not used to fix variables or add bound changes. If all fractionalities are below the feasibility tolerance, then SCIP simply branches on the first fractional variable.

Amongst the separator plugins, currently only the generation of Gomory mixed-integer cuts [50] is available in exact solving mode. As shown in [24], these can be created through numerically safe aggregation of rows, yielding an approximation of an optimal LP tableau row. All MIR rounding operations [78] are also performed in double precision, using safe directed rounding. A custom cut postprocessing routine is used to avoid performance issues in exact LP solving [37].

3.1.6 Safe Dual Proof Analysis and Constraint Propagation

As described in [18], the dual proof analysis from [109] has been made safe and learns exact linear constraints from nodes pruned due to an infeasible or bound exceeding LP relaxation. Similar to the numerically safe Gomory mixed-integer cuts, safe dual proof analysis uses floating-point arithmetic with directed rounding to compute a weighted aggregation of the constraints given by the dual solution of the floating-point LP. Like in floating-point mode, the learned clauses are marked not to be checked nor separated into the LP relaxation, and only propagated. To this end, the exact linear constraint handler is equipped with safe constraint propagation using floating-point arithmetic with directed rounding. This propagation is also applied to all other, model-defining constraints of the exact linear constraint handler. For details and a computational study, see [18].

3.1.7 A Repair Mechanism for Solutions from Floating-Point Heuristics

Although primal heuristics can not provide wrong results as long as the solutions they produce are correctly checked for feasibility, some of SCIP's default heuristic plugins also derive problem reductions, e.g., constraints that express the infeasibility of a set of variable fixings. In SCIP 10.0, all default floating-point heuristics are marked as safe to use in exact solving mode, because all unsafe reductions have been deactivated in exact solving mode. However, the chance that a heuristic that is based on floating-point arithmetic and that uses error tolerances can produce an exactly feasible solution is rather low. Motivated by this, the exact solving mode provides a new constraint handler `exactsol`, which postprocesses candidate solutions from primal heuristics in order to make them exactly feasible. Although a constraint handler, the plugin does not constitute a constraint and is not designed to check solutions for feasibility. Instead, it takes an approximately feasible floating-point solution, rounds and fixes the values of all integer variables, and solves the remaining LP with a call to the exact LP solver. In order to control and save computational effort, candidate solutions can be buffered and only highly promising candidate solutions are processed immediately. Periodically, buffered solutions are processed in order of best approximate objective function value. For a more detailed discussion, see [36].

3.1.8 Certification and Verification

Mathematically proven exactness of an algorithm is clearly not sufficient to ensure that the algorithm is implemented without errors. For a program of vast complexity such as SCIP, it is currently impossible to formally verify the correctness of the implementation. Therefore, SCIP 10.0 provides the option to write a certificate file in the `VIPR` format [22], which can be verified independently of the solving process. While this creates an overhead both for proof logging and proof checking, the overhead does not exceed the time needed for solving. While this does not guarantee correctness of the implementation, it does provide a way to guarantee that the reasoning of the solving process and the optimal solution for a given problem instance are correct.

A certificate file contains the following information:

– the problem statement in exact arithmetic,
– an optimal solution and claimed primal and dual bounds on its objective value,
– the derivation section, which in total certifies optimality of the solution.

The derivation section is a sequential encoding of the branch-and-bound tree that was traversed during the solving process. It is restricted to the following arguments:

1. Conical combinations of inequalities: Given a system of valid linear inequalities $Ax \leq b$, new constraints $c^\top x \leq d$ can be added to the certificate by supplying coefficients $\lambda_i \geq 0$ such that $\sum_i \lambda_i A_i = c$ and $\sum_i \lambda_i b_i = d$, where A_i is the ith row of A. This argument is used to prove dual bounds obtained from LP relaxations, but also for the aggregations used in creating cuts and for dual proof conflicts, cf. Sections 3.1.5 and 3.1.6.
2. Chvátal-Gomory rounding: Given a valid inequality $a^\top x \leq b$ with all coefficients a_i and variables x_i integer, the right-hand side can be rounded to the next integer value $b' = \lfloor b \rfloor$.

3. Disjunction logic: New constraints can be added to the certificate as assumptions, which are then used to prove the conditional validity of other constraints. Eventually, a constraint that is valid given two assumptions that form a disjunction can also be made valid without the assumptions. This "unsplitting" argument is used to derive dual bounds of a parent node from its children and to certify validity of the mixed-integer rounding formula, see [37].

For more technical details on this certificate format, we refer to [22].

Because correctness of the presolving process can currently not be certified in this format, SCIP 10.0 produces the above certificate for the LP-based branch-and-cut process of the transformed problem after presolving. In addition, a certificate is written to allow verifying the primal bound for the original problem by checking feasibility of the final solution in the original problem space.

A C++ proof checker to verify the correctness of these certificate files is available as part of the SCIP Optimization Suite GitHub repository [107]. A more rigorous proof checker formally verified in HOL4 is available through the CakeML framework [69].

3.1.9 Computational Performance of Exact SCIP

In order to analyze the performance of the current state of the exact solving mode, it is insightful to compare it to two different variants of floating-point SCIP: the default setting ("fp-default") and the reduced variant ("fp-reduced"), where all features not available in exact solving mode are disabled. The test was performed on the MIPLIB 2017 benchmark testset [46] with a two hour time limit and three random seeds per instance. All reported times are shifted geometric means with a shift of one; node counts are reported with a shift of 100. We report results both for the subset of instances that could be solved to optimality by at least one setting (one-solved), and for the subset of instances that could be solved to optimality by all settings (both-solved).

The exact setting was able to solve 161 instances to optimality, while the reduced setting was able to solve 235 instances (+46%), and the default setting was able to solve 342 instances (+112%). In terms of solving time, we observe a slowdown factor of 2.6 for the exact solving mode compared to the reduced mode for one-solved and of 3.6 on both-solved. When comparing against the default version of SCIP, that factor is 10.8 for one-solved and 6.8 for both-solved.

It is noteworthy that the number of nodes explored in the exact solving mode is significantly higher than in the reduced mode (+155%), which has several causes such as the fact that strong branching reductions are disabled in exact solving mode, that exactly feasible primal solutions are harder to find, or the requirement to reach a primal dual gap of exactly zero. Overall these results suggest a slowdown factor between 3 and 4 for solving MIPs exactly with the same features and a current slowdown of around 10 times compared to fp-default. Note that this can be vastly greater or smaller depending on the problem instance.

3.2 Presolvers

3.2.1 Implied Integrality Detection using Network Submatrices

Implied integrality, also referred to as implicit integrality, refers to implication of integrality of some variables by integrality of others in combination with the constraints. Already since its inception, SCIP detects implied integer variables using a primal detection method that detects implied integrality for variables with ± 1 coefficients in equations with integral coefficients and right-hand side. In Version 1.1, a dual detection method was added to SCIP, which detects implied integrality for variables that appear in inequalities

Table 5: Performance comparison of exact SCIP against floating-point SCIP with default settings and in a reduced variant, where all features not available in exact solving mode are disabled, taken from [35]. Times are in seconds.

Testset	setting	count	solved	time	(relative)	nodes	(relative)
	exact	350	161	2731.2	–	21109.3	–
one-solved	fp-reduced	350	235	1059.3	0.39	14184.5	0.67
	fp-default	350	342	255.0	0.10	2182.2	0.10
	exact	153	153	798.2	–	8462.8	–
both-solved	fp-reduced	153	153	223.0	0.28	3322.1	0.39
	fp-default	153	153	118.2	0.15	1272.7	0.15

only with ± 1 coefficients. Although both methods have existed in SCIP and other MILP solvers for a long time already, they were only first described in detail in [6]. The authors of [6] conduct experiments with Gurobi and show that enabling implied integer detection reduces the solving time by 13 % and the node count by 18 % on models that take longer than 10 seconds to solve in their internal testing set.

Implied integer variables can be beneficial for performance in several ways. Implied integer variables that are continuous variables in the original model may be used in branching. Furthermore, their integrality can be exploited to derive stronger cut coefficients, most notably in Mixed-Integer Rounding cuts [78], and can enable new classes of cutting planes that require rows with integer variables only. Their integrality can also be used in propagation techniques and conflict analysis to derive stronger bounds. For implied integer variables that are integer in the original model, the redundancy of integrality constraints may be helpful, and can be exploited in primal heuristics such as diving methods or local neighborhood search to limit the number of branching candidates that are to be fixed or explored. Moreover, it is not necessary to branch on implied integer variables within branch-and-bound.

Both the primal and the dual detection method can only detect implied integrality of one variable at a time. Two of the contributors recently investigated implied integrality for MILP problems [105]. A sufficient condition for implied integrality was identified to be integrality of the polyhedron that remains after fixing a subset of integer variables to integer values, independent of the fixed values. To leverage this insight, they formulate a method, called *TU detection*, to detect implied integrality by detecting *totally unimodular* submatrices in the constraint matrix. The primary advantage of this algorithm compared to the primal detection method and the dual detection method is that it can detect implied integrality of more than one variable at a time.

We now present the TU detection algorithm as it was implemented in SCIP 10.0. For the MILP problem as given in (1), it attempts to find a partition of the variable set $\mathcal{N}$ into three pairwise disjoint sets $\mathcal{S}$, $\mathcal{T}$, and $\mathcal{U}$. The set $\mathcal{S}$ must consist of integer variables, while $\mathcal{T}$ and $\mathcal{U}$ may consist of both continuous and integer variables. The TU detection algorithm requires the problem to be of the form

$$\begin{aligned} P = \{(x,y,z) \in \mathbb{Z}^{\mathcal{S}} \times \mathbb{R}^{\mathcal{T}} \times \mathbb{R}^{\mathcal{U}} : Ax + By \le d,\ Ex + Fz \le h,\ \ell \le y \le u, \\ y_i \in \mathbb{Z} \ \text{ for all } i \in \mathcal{T} \cap \mathcal{I},\ z_i \in \mathbb{Z} \ \text{ for all } i \in \mathcal{U} \cap \mathcal{I}\}, \end{aligned}$$

where A and d are integral, the bounds ℓ and u are integral or infinite, and B is totally unimodular. The variable bounds for x and z can be assumed to be implicit in the other constraints. Then, after fixing the x-variables to any integer vector $\bar{x} \in \mathbb{Z}^{\mathcal{S}}$, the remaining problem $Q(\bar{x}) = \{(x, y, z) \in P : x = \bar{x}\}$ can be written as

$$Q(\bar{x}) = \{(x, y, z) \in \mathbb{Z}^{\mathcal{S}} \times \mathbb{R}^{\mathcal{T}} \times \mathbb{R}^{\mathcal{U}} : By \le d - A\bar{x},\ Fz \le h - E\bar{x},\ \ell \le y \le u\ ,\ x = \bar{x},\\ y_i \in \mathbb{Z} \text{ for all } i \in \mathcal{T} \cap \mathcal{I},\ z_i \in \mathbb{Z} \text{ for all } i \in \mathcal{U} \cap \mathcal{I}\}.$$

Note that $Q(\bar{x})$ is the Cartesian product of two disconnected subproblems over y and z (and the fixed vector $\bar{x}$). The linear programming relaxation of the subproblem over y is given by $Q^y(\bar{x}) = \{y \in \mathbb{R}^{\mathcal{T}} : By \le d - A\bar{x},\ \ell \le y \le u\}$. Since B is totally unimodular, the bounds ℓ, u are integral and since the right-hand side $d - A\bar{x}$ is integral, a well-known theorem by Hofmann and Kruskal [56] shows that $Q^y(\bar{x})$ is an integral polyhedron. Provided that the objective considered for $Q^y(\bar{x})$ is linear, there exists an optimal solution $\bar{y}$ of $Q^y(\bar{x})$ that is integral for all $\bar{x} \in \mathbb{Z}^{S}$. Since the y- and z-variables in $Q(\bar{x})$ are disjoint, such a $\bar{y}$ is also integral, feasible and optimal for $Q(\bar{x})$. Furthermore, the feasible solutions of P all lie in $Q(\bar{x})$ for some $\bar{x} \in \mathbb{Z}^{S}$, and each such $Q(\bar{x})$ admits an optimal solution $(\bar{x}, \bar{y}, \bar{z})$ where $\bar{y}$ is integral. This shows that it is safe to assume integrality of the y-variables when optimizing a linear objective over P. In this sense, the x-variables imply the integrality of the y-variables for P. In [105], a more rigorous definition of implied integrality and formal proofs of the TU detection method are presented.

Although totally unimodular matrices can be detected in polynomial time [102], existing implementations of the detection algorithm have high running times [108]. The implied-integer detection algorithm instead only detects *network matrices* and transposed network matrices, which form a large subclass of totally unimodular matrices (see [98, Chapter 19]). In order to detect network matrices, the algorithm uses fast augmentation methods that can determine whether a network matrix can be extended by a new column [16] or row [104]. These methods are then used iteratively to grow a (transposed) network matrix one column at a time.

Note that in the description of P, y and z must be in disconnected blocks, and x must be integral. Thus, the detection method first considers each connected block of the submatrix formed by the continuous columns, and determines if it is network, transposed network or neither, and if it satisfies the given integrality requirements.

After determining implied integrality of the continuous variables, the submatrix B is greedily grown with columns of variables with integrality constraints by using the network matrix augmentation methods.

Some statistics of the new detection method can be found in Table 6. It presents results of the TU detection method on MIPLIB 2017, where 5 instances were excluded due to memory limits and 25 further instances were excluded because they were solved during presolve by both SCIP versions. On average, the new TU detection method is very effective, and detects implied integrality for 18.8 % of the variables, compared to 3.3 % detected using the primal and dual methods. Moreover, the new TU detection method is quite fast and typically runs in a few milliseconds. The changed variable type distribution poses a few challenges. Since (implied) integrality of a variable is a key property of a MIP problem, nearly every algorithm within SCIP uses it, and the performance of SCIP is tuned relative to the variable distribution. Thus, adjusting SCIP to effectively use the detected implied integrality is time- and resource-intensive and requires an ongoing effort. With the release of SCIP 10.0, many important parameters and decisions within the cutting plane and branch-and-bound frameworks have been re-tuned to account for the changed variable type distribution. However, the performance impact of these changes is still only relatively neutral, which is in stark contrast to the results obtained by Gurobi [6]. Consequently, implied integer detection is not enabled by default yet in SCIP 10.0. In future releases, we hope to further improve the exploitation of implied integrality within SCIP.

SCIP 10.0 has several API changes that are closely connected to implied integers. First of all, the implied integer variable type flag is deprecated. Instead, an additional

Table 6: Implied integer detection on 1035 instances of the MIPLIB 2017 collection set [46]. For a set $X \subseteq \mathcal{N}$, X_{int} (X_{con}) indicates the set of variables in X that (do not) have integrality constraints in the original model. $\mathcal{T}$ indicates the set of implied integer variables. The numbers and ratios reported are based on the presolved model, and averaged over all instances tested. The number of variables reported is the shifted geometric mean of the number of variables with shift 10. The detection time reported is the shifted geometric mean with a shift of 1 ms. All other means are arithmetic means.

Method	SCIP 9.2.1	SCIP 10.0.0
affected instances	203	**712**
mean of ratio $\frac{\lvert\mathcal{T}\rvert}{\lvert\mathcal{N}\rvert}$	3.3 %	18.8 %
mean of ratio $\frac{\lvert\mathcal{T}_{\text{con}}\rvert}{\lvert\mathcal{N}\rvert}$	3.2 %	7.5 %
mean of ratio $\frac{\lvert\mathcal{T}_{\text{int}}\rvert}{\lvert\mathcal{N}\rvert}$	0.1 %	11.3 %
mean of ratio $\frac{\lvert\mathcal{N}_{\text{con}}\setminus\mathcal{T}\rvert}{\lvert\mathcal{N}\rvert}$	26.5 %	22.2 %
mean of ratio $\frac{\lvert\mathcal{N}_{\text{int}}\setminus\mathcal{T}\rvert}{\lvert\mathcal{N}\rvert}$	70.2 %	59.0 %
number of variables	6937	6944
mean detection time	–	**13 ms**

flag that, separately from the variable type, indicates whether a variable is implied integer or not. The new flag can take three values; one for no implied integrality, one for implied integrality such as derived by the TU detection method and dual detection method, and one for the implied integrality detected by the primal detection method. The latter is distinguished because it guarantees that a variable takes an integral value in every solution after fixing other integers. This strong property can be used by SCIP in some algorithms, most notably when aggregating variables. In contrast, the dual detection method and the TU detection method only guarantee the existence of an integral solution value, but do not enforce it. This may be disadvantageous, as there are cases where the exact integral values of the y variables are desired, such as primal solutions. For example, it may happen that a primal heuristic uses implied integrality and finds a solution with fractional y that otherwise satisfies the integrality constraints. Then, one needs to find a vertex solution of the LP relaxation of $Q^y(\bar{x})$ to recover the integral solution of the y variables, which can be costly to compute. One additional advantage of the new implied integer flag is that it enables running the dual detection method for variables with integrality constraints, which was previously not possible in SCIP.

3.3 Symmetry Handling

A symmetry of an MILP or MINLP transforms feasible solutions into feasible solutions while preserving their objective values. When solving problems containing symmetries, (spatial) branch-and-bound algorithms arguably create symmetric subproblems, which contain equivalent information. Branch-and-bound can thus be accelerated by removing symmetric subproblems, which is the goal of symmetry handling.

Since Version 5.0, many methods for handling so-called permutation symmetries have been added to SCIP. SCIP 10.0 extends many of these methods to reflection symmetries, see below for a definition of both types of symmetries, and introduces new methods for special symmetries. Moreover, SCIP's capabilities to detect symmetries have been extended to problems containing Pseudo-Boolean or disjunctive constraints.

3.3.1 Handling Reflection Symmetries

SCIP can detect and handle two types of symmetries: permutation symmetries and reflection symmetries. A map $\gamma\colon \mathbb{R}^n \to \mathbb{R}^n$ is a *permutation symmetry* of an MILP or MINLP if it defines a symmetry of the respective problem, as defined above, and there exists a permutation π of $[n]$ such that $\gamma(x) = (x_{\pi^{-1}(1)}, \dots, x_{\pi^{-1}(n)})$ for all $x \in \mathbb{R}^n$. That is, permutation symmetries reorder the entries of a variable vector. A symmetry $\rho\colon \mathbb{R}^n \to \mathbb{R}^n$ of an MILP or MINLP is called a *reflection symmetry* if there exists a permutation π of $[n]$ and a vector $s \in \{-1, 1\}^n$ such that $\rho(x) = (s_1 x_{\pi^{-1}(n)}, \dots, s_n x_{\pi^{-1}(n)})$ for all $x \in \mathbb{R}^n$. That is, reflection symmetries reorder the entries of a variable vector like a permutation symmetry, but additionally, entries can be negated (reflected at the origin). Actually, SCIP can detect more general reflection symmetries that reflect variables at a point different than the origin, e.g., binary variables can be reflected at $\frac{1}{2}$ to exchange 0-entries and 1-entries. For the ease of exposition, the presentation is restricted to reflections at the origin though; see [58] for a discussion of the general case. The remainder of this section provides an overview of the symmetry handling methods that have been made compatible with reflection symmetries in SCIP 10.0, and new methods that have been added to SCIP.

SST Cuts Let Γ be a group of permutation symmetries. Liberti and Ostrowski [74] and Salvagnin [97] introduced *SST cuts* as a family of symmetry handling inequalities, which take the form $x_i \geq \gamma(x)_i$ for all $i \in [n]$ and $\gamma \in \Gamma_i$, where $(\Gamma_i)_{i \in [n]}$ is a carefully selected family of subgroups of Γ. In SCIP 10.0, a version of SST cuts has been implemented that replaces the permutation symmetry γ by a reflection symmetry ρ (not necessarily with respect to the origin). That is, the generalized SST cut is defined as $x_i \geq \rho(x)_i$.

Orbitopes In many applications, such as graph coloring [65], the variables of an MILP or MINLP can be arranged in a matrix $X \in \mathbb{R}^{s \times t}$ such that the symmetries can reorder the columns of the matrix arbitrarily. To handle such column symmetries, which are called orbitopal symmetries, a standard approach is to enforce that a solution has lexicographically sorted columns. To achieve this sorting, previous versions of SCIP implemented both symmetry handling inequalities [57, 59] as well as the propagation algorithm orbitopal reduction [13, 34]. Moreover, if the matrix X consists of binary variables and every feasible solution has at most or exactly one 1-entry per row, specialized cutting planes [65] and propagation algorithms [66] can be used.

In SCIP 10.0, the methods for orbitopes have been extended by also taking reflection symmetries into account. When SCIP detects orbitopal symmetries for a variable matrix X, it is now also checked whether there is a reflection symmetry that simultaneously reflects all variables within a single column of X. If this check evaluates positively, by symmetry of all columns of X, every column admits such a reflection. Hence, by possibly applying these reflection symmetries, one can always guarantee that there is a feasible solution in which all entries in the first row of X are nonnegative. The nonnegativity of the first row is then enforced by adding the inequalities $X_{1,j} \geq 0$ for all $j \in [t]$.

Double-Lex Matrices In applications like the disk packing problem [101], richer symmetries than orbitopal symmetries arise. The goal of one of this problem's variants is to find the maximum value r such that n nonoverlapping disks of radius r can be packed into $[-1, 1]^2$. This problem can be modeled by introducing a variable for the radius and a variable matrix $X \in [-1, 1]^{n \times 2}$, in which entry $X_{i,j}$ models the j-th coordinate of the center of disk i. Given a solution matrix $\bar{X}$, equivalent solutions can be found by permuting the rows of $\bar{X}$ arbitrarily (all disks are identical). Moreover, since the disks are packed into a square, the order of the columns of $\bar{X}$ can be exchanged, and each individual column can be reflected. That is, there are orbitopal symmetries for both exchanges of columns and rows, and columns can be reflected individually.

In SCIP 10.0, a heuristic has been added to detect a generalization of such symmetries. This heuristic tries to identify variable matrices with block shape

$$B = \begin{pmatrix} B_{1,1} & \dots & B_{1,t} \\ \vdots & \ddots & \vdots \\ B_{s,1} & \dots & B_{s,t} \end{pmatrix},$$

where each block $B_{i,j}$ is a submatrix of compatible dimensions and there are orbitopal symmetries on the row and column blocks of this matrix. That is, for each row and column block

$$B_{i,\cdot} = (B_{i,1}, \dots, B_{i,t}),\ i \in [s],\ \text{and } B_{\cdot,j} = (B_{1,j}^\top, \dots, B_{s,j}^\top)^\top,\ j \in [t],$$

respectively, there is an orbitopal symmetry on the rows of $B_{i,\cdot}$ and columns of $B_{\cdot,j}$. The row and column symmetries of the disk packing problem arise then as a special case, in which $s = t = 1$.

To handle orbitopal symmetries of row and column blocks, SCIP applies orbitopal reduction on the submatrices $B_{i,\cdot}$ and $B_{\cdot,j}$, respectively. That is, both the rows and columns can be sorted lexicographically, leading to a so-called double-lex matrix. For the special case $s = t = 1$, SCIP additionally checks whether there exists a reflection symmetry that simultaneously reflects all variables within a single column. If this is the case, due to the presence of row symmetries, one can assume that the first half of the variables in the first column only take nonnegative values; see Khajavirad [68]. Moreover, some entries of columns $2, \dots, t$ can be assumed to be nonnegative, too; see [58]. This is enforced by adding the inequalities $X_{i,j} \geq 0$ for the nonnegative entries (i, j).

Simple Symmetry Handling Inequality Suppose there exists a reflection symmetry that simultaneously reflects all variables of a problem, but does not change the order of the variables, i.e., the permutation π of this reflection is the identity. This symmetry can be handled by the simple inequality $\sum_{i=1}^{n} x_i \geq 0$; see [58]. Because this inequality is rather weak, it is only applied when SCIP does not detect that any other symmetry handling method seems to be useful.

3.3.2 Extensions of Symmetry Detection

Symmetries of an MILP or MINLP are usually detected by computing automorphisms of a suitably defined graph; see Salvagnin [96] and Liberti [73]. Since SCIP 9.0, symmetry detection graphs are created via an optional callback for constraint handlers; see [58] for a detailed discussion. Symmetries of a problem can then be detected if all active constraints handlers implement this callback.

In SCIP 10.0, these symmetry detection callbacks have been implemented for Pseudo-Boolean and disjunctive constraints, thus extending the range of problems for which SCIP can detect symmetries.

3.3.3 Further Enhancements

To compute automorphisms of symmetry detection graphs, previous versions of SCIP provide interfaces to the external graph automorphism software BLISS [63, 64] and NAUTY [80]. The release of SCIP 10.0 adds another interface to the software DEJAVU [7, 8, 9]. In contrast to previous versions, BLISS is no longer shipped with SCIP; instead, NAUTY is provided with the SCIP release. Since we observed that the software NAUTY can spend a substantial amount of time on computing automorphisms, a work limit has been introduced to terminate NAUTY prematurely. This limit can be adapted via the parameter `propagating/symmetry/nautymaxlevel`, which controls the maximum depth level of NAUTY's search tree.

Furthermore, some parts of the code have been refactored. Previous versions of SCIP contained the constraint handler plugin `cons_orbitope` to handle both plain orbitopes and those incorporating set packing and partitioning constraints on the rows. To save memory, SCIP 10.0 contains two new plugins `cons_orbitope_full` and `cons_orbitope_pp` to handle the different types of orbitopes, respectively. To ensure backward compatibility, the old plugin still exists and serves as an interface to the two new ones, i.e., the old API can still be used to add orbitope constraints.

3.4 Cut-based Conflict Analysis

In SCIP, conflict analysis is applied whenever a local infeasible constraint is detected at a node of the branch-and-bound tree. A common source of such detections is propagation, which may occur after multiple rounds on the current node. The goal of conflict analysis is to "learn" a constraint that would have revealed the infeasibility earlier. Such a constraint would have propagated at an earlier node, preventing SCIP from visiting the infeasible node in the first place. Up to SCIP 9.0, infeasibilities were analyzed with a SAT-inspired, graph-based conflict analysis [2]. This approach represents the sequence of branching decisions and propagated deductions that led to the contradiction as a directed acyclic graph. From this graph, a valid constraint can be inferred by identifying a subset of bound changes that separates the decisions from the node where the contradiction was reached. The learned constraints are disjunctive constraints, are implied by the original problem, and are therefore globally valid.

As of SCIP 10.0, cut-based conflict analysis was introduced [82]. The method is inspired by conflict-analysis techniques from Pseudo-Boolean solvers [21, 71, 38, 81] and operates directly on more expressive linear inequalities rather than conflict graphs. It proceeds via linear combinations, integer roundings, and cut generation. During branch-and-bound, each branching decision or propagation tightens the local domain L_T. If propagation detects infeasibility (i.e., when propagating a constraint $C_{\texttt{conf}}$ on the local domain L_T), conflict analysis is triggered. In this context, $C_{\texttt{conf}}$ is the conflict constraint, while the constraint $C_{\texttt{reason}}$ that defined L_T (that is, $C_{\texttt{reason}}$ propagated under the previous local domain L_{T-1}) is the reason constraint. Together, L_{T-1}, $C_{\texttt{reason}}$, and $C_{\texttt{conf}}$ form an infeasible system. Each iteration aims to derive a single constraint C that is globally valid and infeasible in L_{T-1}. Once such a constraint C is identified, it becomes the new conflict constraint $C_{\texttt{conf}}$. In each iteration, the reason constraint $C_{\texttt{reason}}$ is set to the constraint that propagated the most recent bound change contributing to the infeasibility of $C_{\texttt{conf}}$ and the algorithm continues until a globally valid constraint is obtained that could have prevented the infeasibility before the latest branching.

In an iteration of conflict analysis, it may not be possible to derive a globally valid conflict constraint C using only the inequalities $C_{\texttt{reason}}$ and $C_{\texttt{conf}}$. Figure 1 illustrates this situation. The left plot shows the region defined by the reason constraint $C_{\texttt{reason}}$, the conflict constraint $C_{\texttt{conf}}$, and the global domain. The middle plot shows the mixed-integer hull of this region, where `x1` is the only integer variable. The right plot shows the intersection of the mixed-integer hull with the local domain. In particular, the leftmost and rightmost red points are integer-feasible, while the middle red point is a convex combination of them that intersects the local domain. Because the local domain intersects the mixed-integer hull, no globally valid inequality can be derived that separates the local domain from the hull. In such cases, the reason constraint $C_{\texttt{reason}}$ needs to be modified to ensure that the new conflict constraint $C_{\texttt{conf}}$ remains infeasible in the previous local domain. This is done by applying so-called *reduction rules* to the reason constraint that can be activated by setting the parameter `conflict/reduction` to `m` (which stands for `MIR` reduction) and, for the case of mixed-binary constraints, set `conflict/mbreduction`

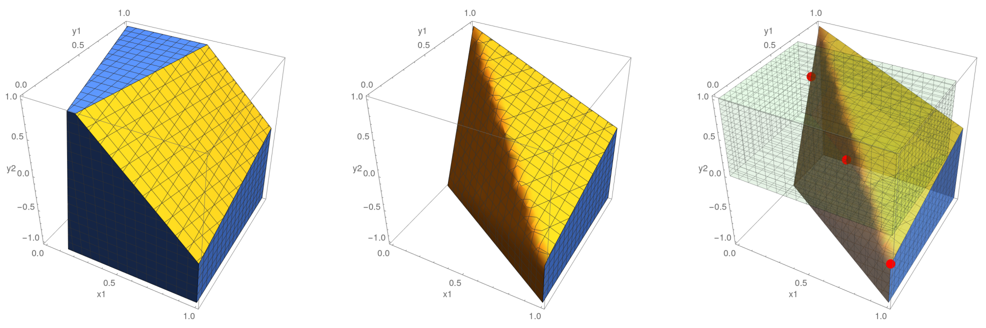

Figure 1: Left: Region defined by $C_{\texttt{reason}}$, $C_{\texttt{conf}}$, and the global domain. Middle: Mixed-integer hull, where `x1` is the only integer variable. Right: Mixed-integer hull intersected with the local domain. In particular, two integer-feasible points (leftmost and rightmost red points) and a convex combination of them that intersects the local domain (middle red point).

to `true`. To activate cut-based conflict analysis, the parameter `conflict/usegenres` has to be set to `true`.

3.5 Separators

A new separator `sepa_flower` dealing with products of nonnegative variables was added to SCIP. It works with logical AND constraints $z = x_1 \cdot x_2 \cdot \ldots \cdot x_\ell$ for binary variables $x_i \in \{0, 1\}$ and products appearing in nonlinear constraints. In the former, z is called *resultant* and the x_i are called *operands*. The purpose of the new separator is to generate cutting planes that strengthen the LP relaxation in the presence of multiple such constraints. The collection of all constraints can be represented via the *multilinear hypergraph* $G = (V, E)$, which has a node $v \in V$ per variable x_v that appears as an operand in at least one AND constraint and a hyperedge $e \in E$ per constraint, where the resultant variable satisfies $z_e = \prod_{v \in e} x_v$.

This hypergraph is constructed after presolving the problem by inspecting all AND constraints and scanning expression trees of nonlinear constraints. Moreover, *overlap sets*, which are sets of the form $e \cap f$ for e, $f \in E$, are gathered using hash tables, from which *compressed sparse row* and *column* representations of the incidences among hyperedges, overlap sets and nodes are computed.

While there exist many classes of valid inequalities [26, 28, 29, 27, 30], we restricted the current implementation to *k-flower inequalities* [29] for $k = 1$, 2, since the separation problem can be solved very efficiently once the overlap sets are available. Such an inequality is parameterized by one *center* edge $e \in E$ and k *neighbor edges* $f_1, f_2, \ldots, f_k \in E$, all of which must intersect e. It reads

$$z_e + \sum_{i=1}^{k} (1 - z_{f_i}) + \sum_{v \in R} (1 - x_v) \geq 1,$$

where $R \coloneqq e \setminus \bigcup_{i=1}^{k} f_i$ denotes the nodes of e that are contained in no neighbor edge. By simple enumeration, the separation problem can easily be solved in time $\mathcal{O}(|E|^{k+1})$. Under the reasonable assumption that the size $|e|$ of every edge is bounded by a constant, this can be reduced to $\mathcal{O}(|E|^2)$ as done in the implementation [31]. By exploiting overlap sets using the implemented data structures, this is reduced to $\mathcal{O}(|E|)$.

Hyperedges for the new separator can, in principle, also be derived from product expressions involving continuous variables. However, this requires nonnegative lower bounds that are relaxed to 0, and in our tests we observed a performance loss, which is why this is disabled by default. Users can enable this feature via the parameter `separating/flower/scanproduct`.

Table 7: Performance comparison for instances from MILP, MINLP and Pseudo-Boolean testsets. MILP and Pseudo-Boolean experiments were run using SoPlex 8.0.0 as an LP solver and those for MINLP using CPLEX 12.10.0.0. Handling products of continuous variables remained disabled for all experiments.

			Without `sepa_flower`			With `sepa_flower`			relative	
Testset	subset	instances	solved	time	nodes	solved	time	nodes	time	nodes
MILP	all	698	604	291.7	2815	605	290.7	2810	1.00	1.00
MILP	affected	14	13	369.4	261	14	313.2	223	0.85	0.85
MINLP	all	822	810	18.9	2407	810	18.7	2377	0.99	0.99
MINLP	affected	21	21	8.6	1712	21	5.1	1023	0.60	0.60
Pseudo-Boolean	all	450	423	7.8	333	423	7.7	331	1.00	0.99
Pseudo-Boolean	affected	11	11	12.0	359	11	9.7	258	0.81	0.72

Computational results are shown in Table 7 for MILPs, MINLPs and Pseudo-Boolean instances, which give a clear advantage on affected instances although there are not too many. However, thanks to the very efficient computation of the hypergraph G, the separator is performance-neutral in case multilinear structures are absent.

3.6 Heuristics

3.6.1 Kernel Search

In SCIP 10.0 the Decomposition Kernel Search (`DKS`) heuristic is introduced. This includes the basic framework of Kernel Search (KS) explained in the following. The extensions of KS to respect additional decomposition information is described in the next subsection. Originally, the heuristic was introduced in [10, 11] to leverage a substructure on binary variables, which occurs in certain applications. It was later refined to integrate general integer variables into its functionality [51], which serves as basis for the present implementation of KS, see [53] for details.

In KS, a subset of the integer variables is defined as the *kernel* $\mathcal{K}$. It is supposed to contain variables that are likely to be active in a solution. Here and in the remainder of the section, a variable is called *active* in a solution when its value is nonzero and not at the lower bound. In the opposite case, a variable is called *inactive*. Note that this definition of (in)activity is a generalization to MILP problems of the definition in the classical literature, where typically only lower bounds of zero are considered. One exemplary way to identify such a property is to consider active variables in the LP solution. The remaining integer variables $\mathcal{I} \setminus \mathcal{K}$ are divided into *buckets*. In the core procedure of KS, the kernel is united with a bucket $\mathcal{B}$ and Problem (1) is restricted to this union. The restriction is obtained by fixing every integer variable not in the union to be truly inactive, in particular, to the lower bound, if it is finite, or to zero otherwise. Note that variables without a finite lower bound but with a negative upper bound are considered active in the current implementation and thus remain in the kernel.

This identification of kernel and buckets as well as their handling is also performed for continuous variables, which is in contrast to the original sources but is based on the extensions below. To force a binary or general integer variable from the bucket to be not at its lower bound and thus potentially be active, the parameter `heuristics/dks/advanced/addUseConss` controls the addition of the constraint

$$\sum_{i \in \mathcal{K} \cup \mathcal{B}} x_i \geq \sum_{i \in \mathcal{K} \cup \mathcal{B}} \ell_i + 1.$$

Here, only variables with finite lower bounds ℓ_i are considered. After solving the restricted problem, the kernel is updated by adding active variables from the current bucket and deleting inactive kernel variables with respect to the obtained solution. The update is performed whenever an improving solution is found. This procedure is iterated until solutions for all kernel-bucket combinations are computed or resource limits are reached.

In [53], several extensions to the original work are introduced. There may already exist a feasible solution to the original problem when the heuristic is called. This can be exploited by using the parameter `heuristics/dks/usebestsol` to define the initial kernel $\mathcal{K}$ by considering the active variables in the currently best solution. The remaining inactive variables are typically assigned to distinct buckets of similar size, after being sorted by their reduced cost in the LP relaxation. However, the reduced costs may be distributed in different orders of magnitude. This motivates an assignment to buckets of different size, yet based on uniform ranges of the logarithm of the reduced costs. The parameter `heuristics/dks/redcostlogsort` is provided to switch between these two approaches.

When solving the sequence of kernel-bucket-subproblems, the trade-off between solving times and objective improvements must be controlled carefully. To do this, the gap limit for each of these subproblems is chosen adaptively based on whether the node limit of the previously solved problem is reached.

Two points are noteworthy with regard to the algorithmic scheme described: First of all, there is an initial solve restricted to the kernel only. Second, although it was introduced and defined for MILPs, the above is no restriction for MINLPs in theory. However, extending the implementation to this general problem class requires special care.

3.6.2 Decomposition Kernel Search

Halbig et al. [53] extend KS explained in the previous section by the possibility to utilize decomposition information, called Decomposition Kernel Search (`DKS`). Details on the representation of decomposition information can be found in the release report for SCIP 7.0 [43, Section 4.2]. Such decomposition information indicates relations among variables by specifying a block structure of the constraint matrix A in (1).

Since empirical tests yielded active variables in MILP-solutions for variables across all blocks and variable types, this is taken into account when creating the kernel and the buckets. For this reason, kernel and buckets are determined for each block and variable type and are then united. Since this strategy is performed across several levels, in particular for the blocks and variable types, the resulting structure of the final kernel and buckets is referred to as *multi-level.*

There are different types of decomposition information depending on how the information was generated, e.g., through graph partitioning or flow modeling. Computational experiments on instances associated with decompositions do not show a favor towards a special type of decomposition information but for specific problem classes. In particular, `DKS` has a positive effect on solving binary problems, which KS was originally presented for. On top of that, the solving times for problems with binary and integer variables without continuous variables are reduced. The effect is attributed to the multi-level structure, as it seems to appropriately capture the occurrence of active variables in a solution. Since it has a positive impact for certain problem types, but requires finetuning to be applicable in general, the heuristic is disabled by default. For a detailed description please refer to [53].

3.7 Branching Rules

3.7.1 Ancestral Pseudocosts

Classical pseudocosts [14] estimate the immediate LP relaxation improvement from branching but ignore downstream effects. Our proposed approach, ancestral pseudocosts, aims to approximate this longer-term influence by aggregating improvements from subsequent levels, weighted by a discount factor. The following example illustrates how these updates are recorded during branching.

Assume we are at a node N_0 with LP relaxation objective value $L_0 \in \mathbb{R}$. Consider a candidate variable x with LP value $x_0^{lp} \in \mathbb{R}$. When we branch on $x \leq \lfloor x_0^{lp} \rfloor$, we create a new node N_1 with LP relaxation objective value $L_1 \in \mathbb{R}$. At node N_1, we update the traditional pseudocost $PS_0(x)$ for the variable x by adding a record $(L_1 - L_0)/\{x_0^{lp}\}$, where $\{x_0^{lp}\} := x_0^{lp} - \lfloor x_0^{lp} \rfloor$. Now consider the future-case where N_1 is the considered node and y the candidate variable with LP solution value $y_1^{lp} \in \mathbb{R}$. When we branch on $y \leq \lfloor y_1^{lp} \rfloor$, a new node N_2 is created with LP relaxation objective value $L_2 \in \mathbb{R}$. At N_2, we now update two records instead of one:

1. The traditional pseudocost $PS_0(y)$ of variable y by adding datapoint $(L_2 - L_1)/\{y_1^{lp}\}$, where $\{y_1^{lp}\} := y_1^{lp} - \lfloor y_1^{lp} \rfloor$.
2. First-level pseudocost $PS_1(x)$ of variable x by adding datapoint $(L_2 - L_1)/\{x_0^{lp}\}$.

In this example, part of the improvement in the LP relaxation bound at node N_2 is attributed not only to the variable y, which was branched on at N_1, but also to the earlier decision to branch on variable x at N_0. This attribution is made through the first-level pseudocost $PS_1(x)$, which receives a discounted share of the bound improvement from L_1 to L_2. In general, for a variable x, we can maintain multiple levels of pseudocosts $PS_k(x)$, where k denotes the number of levels between the branching decision on x and the resulting node where the LP improvement is observed.

These contributions are aggregated using a geometric discount factor $\gamma \in (0, 1]$, leading to the following formulation for the *ancestral pseudocost* of variable x:

$$APS(x) = PS_0(x) + \gamma PS_1(x) + \gamma^2 PS_2(x) + \dots \tag{3}$$

Instead of selecting a branching variable based solely on traditional pseudocosts $PS_0(x)$, we compare variables using their ancestral pseudocosts $APS(x)$. This encourages the solver to favor decisions that not only offer immediate LP bound improvements, but also lead to stronger relaxations deeper in the search tree.

Currently, ancestral pseudocosts are implemented using one-level. Such an implementation can be interpreted as an approximation of lookahead branching [45]. While traditional pseudocosts estimate the effect of strong branching at the immediate child node, incorporating one-level ancestral pseudocosts mimics lookahead branching by capturing downstream LP relaxation improvements, especially when the discount factor $\gamma = 1$. For ancestral pseudocosts to work well, both levels of pseudocosts—$PS_0(x)$ and $PS_1(x)$—must be reliable. In SCIP, we adopt a conservative strategy: if either level is deemed unreliable for any candidate variable, the branching rule falls back to the default (hybrid branching [5] using only $PS_0(x)$) at that node.

To control the discount factor γ applied to ancestral pseudocosts for the "pscost" and "relpscost" branching rules, the parameters `branching/pscost/discountfactor` and `branching/relpscost/discountfactor` have been added to SCIP, respectively (default = 0.2). An additional flag, `branching/collectancpscost`, enables or disables the recording of ancestral pseudocosts (default = "false"). In the default settings of SCIP 10.0, ancestral pseudocosts are disabled until conclusive evidence of improvement is found for general MILPs.

3.7.2 Probabilistic Lookahead for Strong Branching

The hybrid branching rule that SCIP uses by default relies on strong branching calls to select branching variables, especially early in the tree before the pseudocosts of many variables are well initialized and considered reliable. Strong branching is, however, expensive and careful working limits were added to avoid solving a prohibitive amount of LPs. One such working limit is the so-called lookahead number, i.e., the number of consecutive variables that are evaluated without improvement to the dual bound estimate, after which we stop evaluating more candidates. This maximum number is typically static, and a new dynamic criterion was introduced by Mexi et al. [83], based on an abstract probabilistic model of the branch-and-bound tree. Importantly, experiments showed that the new dynamic criterion achieves a reduction in both runtime and tree size. It reallocates strong branching calls at nodes where it is needed while stopping the candidate evaluation early when the probabilistic model estimates a low probability of finding a better candidate than the incumbent one. The new criterion can be activated with the `branching/relpscost/dynamiclookahead` parameter.

3.7.3 Mix Branching on Integer and Nonlinear Variables

When solving problems with nonlinear nonconvex constraints, the constraint handler `cons_nonlinear` would previously consider branching on a variable in a nonlinear nonconvex term only if there were no integer variables with fractional value in the solution to enforce. Such a strict preference for integer variables did not seem appropriate, but it was also unclear when branching on a nonlinear variable was preferable. To support ongoing investigations in this topic, with SCIP 9.1.0 the enforcement priority of `cons_nonlinear` has been increased to be above that of `cons_integral`. Additionally, the option `constraints/nonlinear/branching/mixfractional` has been added to enable considering nonlinear variables ahead of integer variables with fractional value for branching. The new option specifies the minimal average pseudocost count across integer variables that is required to consider branching on a nonlinear variable before branching on a fractional integer variable. By default the option is set to infinity, but when set to a small value, e.g., 0, 1, or 2, fractional integer and nonlinear variables are jointly considered for branching by `cons_nonlinear` as soon as sufficiently many pseudocosts have been computed for integer variables. For this purpose, the scoring of branching variables in `cons_nonlinear` [15, Section 4.2.12] has been extended to consider the fractionality of an integer variable as a score similar to the violation score that is computed for a nonlinear variable. If the scoring on this joint set picks a fractional integer variable however, then the regular branching rules for integer variables are employed.

3.8 Benders' Decomposition

The Benders' decomposition framework [77] has received some small feature updates for SCIP 10.0. First is the introduction of objective types for the subproblems. This aims to allow more flexibility to the user in formulating problems when using the Benders' decomposition framework. Second is the improved identification and handling of master linking variables. The latter feature is designed to enable the integer and no-good cuts on a broader range of problem types. Finally, the decomposition process when supplying a DEC file has been updated so that the original solution is now returned from the Benders' decomposition solve.

3.8.1 Objective type

Classically, the application of Benders' decomposition results in the inclusion of an auxiliary variable to provide an underestimation of the subproblem objective value. If the subproblem can be separated into disjoint problems, then this auxiliary variable can be substituted with the sum of auxiliary variables (one for each subproblem). Prior to SCIP 10.0, the Benders' decomposition framework only supported this classical handling of the subproblem objective.

While the summation of auxiliary variables is theoretically possible for all applications of Benders' decomposition, there are problems where an alternative objective may be beneficial. An example is a multiple machine scheduling problem with a makespan objective. In this case, an application of Benders' decomposition typically involves splitting the subproblems by machine. Then in the master problem, a makespan variable is defined, which is then constrained to take the maximum of the makespan from each individual subproblem. In such a setting, it is more convenient to define the subproblem objective as the minimum of the maximum makespan across all subproblem, compared to some summation objective.

In this release, the Benders' decomposition framework has been extended to support two different objective types: the classical summation and the minimum of the maximum subproblem auxiliary variables. The objective type can be set using the method `SCIPsetBendersObjectiveType()`. The objective type must be set during the problem creation stage, since this setting has an impact on the problem being solved. Note that the different objective types only have an impact if more than one subproblem is used in the Benders' decomposition.

3.8.2 Handling of Master Linking Variables

The linking master variables are a critical part of the Benders' decomposition algorithm. In the SCIP implementation, the linking variables are identified by the existence of a pair of variables with the same name in the master and subproblem. It is assumed that all master variables could potential be linking variables. This assumption led to the incorrect disabling of the integer and no-good cuts.

Prior to SCIP 10.0, the integer and no-good cuts would only be applied if the master problem was pure binary. However, it is possible to apply these cuts when the master problem is an MIP, provided that the linking variables are all binary. This can treated in a more fine-grained manner, where the integer and no-good cuts can be applied to specific subproblems, because these only have binary linking variables.

To facilitate the better handling of integer and no-good cuts, the linking master variables are identified and stored during the initialization stage of the Benders' decomposition framework. Statistics on the number of integer and binary linking variables are recorded for each subproblem. These statistics are then used to determine whether the integer and no-good cuts can be enabled for the subproblems.

3.8.3 Returning Original Problem Solution

One method for applying Benders' decomposition using SCIP is to supply a DEC file specifying the decomposition for an instance. Additionally setting the parameters `decomposition/benderslabels` and `decomposition/applybenders` will decompose the instance and apply Benders' decomposition to solve the problem. In previous versions of SCIP, only the variable values for the master problem solution were directly available. The subproblem solutions could be retrieved by calling `display subsolution <subproblem-index>` from the interactive shell or by calling `SCIPbendersSubproblem` (to retrieve the subproblem), `SCIPsetupSubproblem` (to apply the best known master problem solution), and then `SCIPsolveBendersSubproblem` (to solve the subproblem from scratch). While this approach for retrieving the original problem solution is still necessary when an instance is supplied in its decomposed form, such as using the SMPS instance format or using a custom Benders' decomposition implementation, it was not practical when the original problem instance is known.

The application of Benders' decomposition when supplying a DEC file has been redeveloped in SCIP 10.0. A relaxator has been added to the default plugins, which handles the decomposition and solving of an instance using Benders' decomposition. The motivation for using a relaxator is so that the original instance remains untouched, while the decomposition can be applied within the relaxator. The relaxator acts as a sandbox where it is possible to make the appropriate, destructive modifications to the problems in order execute the Benders' decomposition algorithm.

When the solving process is started in SCIP, the original SCIP instance will execute presolving. During the processing of the root node from the original problem, the Benders' decomposition relaxator will be called first. The Benders' decomposition algorithm attempts to solve the problem to optimality. At the completion of the Benders' decomposition algorithm, the best found primal and dual bounds are returned to the original SCIP instance. If possible, the solution from the decomposed problem is mapped back to the original instance variables.

By default, the original SCIP instance will terminate with an optimal solution or infeasible status. If a limit has been set, a time, gap, or bound limit status could be returned, or a "user interrupt" indicated. If the Benders' decomposition algorithm fails to solve the instance—due to reaching working limits—or the solution could not be returned, a "user interrupt" is triggered. To continue solving the original SCIP instance after the conclusion of the Benders' decomposition algorithm, parameter `relaxing/benders/continueorig` can be set to TRUE.

3.9 IIS Finder

It is a common issue for integer programming practitioners to (unexpectedly) encounter infeasible problems. Often it is desirable to better understand exactly why the problem is infeasible. Was it an error in the input data, was the underlying formulation incorrect, or was the model simply infeasible by construction? A common tool for helping diagnose the reason for infeasibility is an Irreducible Infeasible Subsystem (IIS) finder. An IIS is a subset of constraints and variable bounds from the original problem that when considered together remain infeasible and cannot be further reduced without the subsystem becoming feasible.

Practitioners can use IIS finders, since they narrow their focus onto a smaller, more manageable problem. Note, however, that there are potentially many different IISs for a given infeasible problem, and that IIS finders generally provide no guarantee of an IIS of minimum size. For a complete overview of IISs, see [23], and for an alternate reference detailing the implementation of an IIS finder, see [91].

In SCIP 10.0, users now have the functionality to generate an IIS for any infeasible CIP. This is achieved by calling `SCIPgenerateIIS` via the API or `iis` via the CLI, with the resultant IIS accessible afterwards by calling `SCIPgetIIS` via the API or both `write/iis` and `display/iis` via the CLI. Moreover, users now have the ability to easily develop their own algorithms for generating an IIS, where we call these algorithms *IIS finders*. These have been added as a new SCIP plugin, aptly named `SCIP_IISFINDER`, and therefore a user need only implement the required callbacks to write their own IIS algorithms. Currently, SCIP has implemented the additive and deletion based methods from [52]. The resultant IIS finder containing the additive and deletion based methods is called *greedy*. Many parameters are also made available to the user, with two examples

Table 8: Performance comparison vs Ipopt

		SCIP + CONOPT			SCIP + Ipopt			relative	
Subset	instances	solved	time	nodes	solved	time	nodes	time	nodes
all	812	803	18.8	2526	800	18.2	2361	1.04	1.07
affected	523	520	18.2	3978	517	17.3	3587	1.06	1.11
[10,timelim]	426	423	81.5	8541	420	77.8	8132	1.05	1.05
[100,timelim]	187	184	338.3	43711	181	354.1	43976	0.96	0.99
[1000,timelim]	49	46	703.3	140280	43	1167.9	190723	0.60	0.74
diff-timeouts	9	6	31.4	2009	3	2718.6	42060	0.01	0.05
both-solved	797	797	18.0	2419	797	16.4	2180	1.09	1.11
continuous	184	180	11.2	4033	177	13.1	4460	0.86	0.90
integer	623	618	21.8	2225	618	20.1	1995	1.08	1.12

being: (1) Allowing only constraint deletion and leaving the original set of bounds as is. (2) Allowing (potentially faster) generation of an infeasible subsystem with no guarantee of irreducibility.

3.10 CONOPT Interface

When solving MINLPs, SCIP makes use of local NLP solvers for several purposes. Primal heuristics solve NLPs, obtained by fixing integer variables, instead of LPs in order to find feasible solutions, convex NLP relaxations are used in cut generation and bound propagation, the Benders algorithm makes use of NLP subproblems, and some constraint handlers solve specialized NLPs.

This release introduces an interface to CONOPT (`conopt.com`), a nonlinear programming solver implementing a feasible path algorithm [88] that is based on active set methods [49, 44]. The solver is particularly suited for large, sparse models and settings where a series of mostly similar problems need to be solved, the solution of one problem providing a good starting point for the next.

Table 8 compares the performance of SCIP when using Ipopt and CONOPT as the nonlinear programming solver. The experiments were conducted on a subset of the MINLPLib instances which can be solved by SCIP 8.0.0 under one hour. The time limit was one hour. The results show that using CONOPT instead of Ipopt increases the mean time by 7 % and the mean number of nodes by 4 %. However, on the subsets [100,timelim] and [1000,timelim] it reduces the mean time by 4 % and 40 %, respectively, and the mean number of nodes by 1 % and 26 %, respectively, and the overall number of solved instances is larger by 3 with CONOPT. We observed that the largest differences in performance occurred when one NLP solver was capable of solving an NLP problem and the other reached a time or iteration limit or arrived at a locally infeasible point.

3.11 Technical Improvements

3.11.1 Checking bounds on aggregated variables

In some situations, SCIP terminates with a solution that is feasible in the presolved problem, but has small infeasibilities (above the feasibility tolerance) in the original problem. One possible source for such a situation are aggregated or multiaggregated variables. For example, if a linear constraint $z = x + y$ with $x, y, z \geq 0$ is removed and z is replaced by $x + y$ in the objective function and any other constraint, then a solution with $x = y = -10^{-6}$ can be feasible in the presolved problem (assuming default `numerics/feastol=1e-6`), but $z = -2 \cdot 10^{-6}$ violates the lower bound of z by more than 10^{-6}.

Since SCIP 9.1.0, a new constraint handler `cons_fixedvar` is available that checks whether the bounds of aggregated variables are satisfied in a given solution. If this is not the case, but the solutions needs to be enforced, then the aggregation is added as a cut to the LP relaxation. In the given example, $z = x + y$ would be added. If a cut could not be generated for some reason, then the feasibility tolerance for the LP relaxation is reduced for the current node.

3.11.2 SCIP Statistics Serialization

SCIP 10.0 introduces a new callback method for the table plugin used to write out statistics tables, called `TABLE_COLLECT` that collects the data in the newly introduced `DATATREE` object that represents generic serialization of data. This allows to generate the statistics in a generic way and to write them in different file formats. This new callback method is implemented for all the statistics tables in SCIP, and they can currently be exported to a JSON file using the `SCIPprintStatisticsJSON` function through the C API or through the command line if the written statistics file has a JSON extension.

3.11.3 Writing AMPL `.nl` files

The reader for AMPL `.nl` files has been extended by writing capabilities. It utilizes the NL writer of the `ampl/mp` project and currently supports general and specialized linear constraints and nonlinear constraints.

4 The GCG Decomposition Solver

GCG extends SCIP by providing the functionality of a decomposition-based solver. That is, GCG reformulates a given MILP according to an automatically detected Benders or Dantzig-Wolfe decomposition, and then solves the reformulation by branch-and-cut or branch-cut-and-price, respectively. For the user, no expertise in these techniques is needed, but for an understanding of the following, we assume that the reader is familiar with decomposition methods. We refer to [32] for a recent reference.

The central design paradigm of GCG for its main use case, branch-cut-and-price based on a Dantzig-Wolfe reformulation, is the synchronized work on two SCIP instances, reflecting the original (user-given) and the master (reformulated) models. This design allows for benefiting from SCIP's functionality on both models, which are intimately linked by theory, and thus both needed for a modern branch-cut-and-price implementation. Most decisions and information available in one tree, like branching, cutting, dual bounds, primal solutions, propagations, etc. are mirrored to the other tree. The (relaxed) Dantzig-Wolfe master model lives in a relaxator of the original SCIP instance and is solved by column generation in every node.

4.1 License

GCG has been open-source since its initial release. In order to harmonize the licensing with SCIP, all contributing code authors have approved to switch from the GNU LGPL to the Apache 2.0 license, starting with GCG 4.0.

4.2 GCG Data Structure

GCG 4.0 introduces a GCG data structure similar to SCIP's main data structure. The main purpose is to simplify the C API as many functions now accept a GCG object instead of a SCIP object. Users do no longer need to check whether they have to pass the original or a master problem to the function. The GCG object stores important and frequently accessed pointers. Among these are pointers to the original problem, the master problems (Dantzig-Wolfe and Benders), the relaxator, and the pricer. This also eliminates the need of storing the same pointers in many user data objects or the need of `SCIPfind*` calls. A GCG object can be created using `GCGcreate` and freed using `GCGfree`. We plan to extend this data structure in future releases.

4.3 Extended Master Constraints

The Dantzig-Wolfe reformulation of an original model $\min\{c^\top x : Ax \geq b, Dx \geq d, x \in \mathbb{Z}^n\}$ works with $X \coloneqq \operatorname{conv}\{x \in \mathbb{Z}^n : Dx \geq d\}$. Denote the finite sets of integer extreme (and potentially some integer interior) points and integer extreme rays of X by $\{x_p\}_{p\in P}$ and $\{x_r\}_{r\in R}$, respectively. The Dantzig-Wolfe (discretized) integer master problem

$$\begin{aligned}
\min \quad & \sum_{p\in P}(c^\top x_p)\lambda_p + \sum_{r\in R}(c^\top x_r)\lambda_r \\
\text{s.t.} \quad & \sum_{p\in P}(Ax_p)\lambda_p + \sum_{r\in R}(Ax_r)\lambda_r \geq b \qquad [\pi] \\
& \sum_{p\in P}\lambda_p = 1 \qquad [\pi_0] \\
& \lambda \in \mathbb{Z}_+^{|P|+|R|}
\end{aligned} \tag{4}$$

is (one option for) an equivalent reformulation of the original model, using (among other conditions) the linking

$$x = \sum_{p\in P} x_p\lambda_p + \sum_{r\in R} x_r\lambda_p \tag{5}$$

between the original x and master λ variables. The π, π_0 in brackets denote the dual variables corresponding to the constraints in the LP relaxation. This relaxation is potentially stronger than that of the original problem. To solve the problem by column generation, it is necessary to repeatedly solve the pricing problem $\min\{(c^\top - \pi^\top A)x - \pi_0 : x \in X\}$, where the objective function expresses the reduced costs of a master variable.

In (5) it can be seen that integrality of λ-variables implies integrality of x-variables, but the converse does not hold. Branching (and cutting) can be carried out on both kinds of variables. A branching decision on a fractional original x-variable translates via (5) to a constraint in the (relaxation of the) master problem (4). Even easier, the branching decision can be enforced by a variable bound in the pricing problem. Therefore, tailored pricing algorithms are usually still applicable after branching. This branching (and cutting) on original variables, sometimes called *robust*, is part of GCG.

In certain situations, in particular when the master problem is *aggregated*, users want to branch on fractional sums of master variables. Branching constraints, that are added to the master problem, are of the following form:

$$\sum_{p\in P} f(p)\lambda_p + \sum_{r\in R} f(r)\lambda_r \geq h, \qquad [\gamma] \tag{6}$$

where, again, γ denotes the corresponding dual variable in the master relaxation. The coefficients $f(j)$, $j \in P \cup R$, are usually binary and represent a well-defined subset of master variables. The Ryan-Foster branching [95] on two rows r and s is a classical example where $f(j) = 1 \iff x_{rj} = x_{sj} = 1$, $j \in P$, that is, both components r and s of vector x_j are nonzero. Ryan-Foster is part of GCG, but many more options are conceivable, see Section 4.3.1 for an example.

Note that cutting planes in the master variables, like clique cuts or Chvàtal-Gomory cuts, are also of the form (6).

The coefficient $f(j)$ needs to be computed in the pricing problem as well. Unlike in robust branching, $f(j)$ *cannot* be stated as linear expressions in the original x-variables already present in the pricing problem *alone*: Instead, we introduce a *coefficient variable* y, one for every branching constraint or cutting plane. This variable is constrained to a domain Y that ensures the correct correspondence of the coefficient $y = f(j)$ and the semantics of the branching decision or cutting plane. The set Y is defined by linear constraints in x, y, and potentially auxiliary (integer) variables z. The z-variables are called *inferred* variables in the code. The necessary modifications in the pricing problem

$$\begin{aligned} \min \quad & \left(c^\top - \pi^\top A\right) x - \gamma y - \pi_0 \\ \text{s.t.} \quad & x \in X \\ & y \in Y(x, z) \end{aligned}$$

may entail a nontrivial additional computational burden and/or destroy the applicability of a tailored pricing algorithm. Such branching rules or cutting planes are therefore also called *non-robust*. They can be stronger than robust branching and cutting.

Concluding, in nonrobust branching or cutting the master constraint (6) *does not* result from a Dantzig-Wolfe reformulation of a constraint in the original model, in contrast to the robust case. In GCG 4.0 we therefore introduce the concept of *extended master constraints* which live *only* in the master problem and therefore do not correspond to an original constraint. We provide a new interface for creating and managing extended master constraints to use within branching rules and separators. This interface is deeply integrated into GCG. The modifications to the pricing problem are automatically and dynamically applied and undone in each node as necessary. Dual value stabilization of extended master constraints is supported. When users develop their own master branching rules, it suffices to create extended master constraints using the new interface in GCG 4.0. Moreover, basic functionality for separators is available. However, this is an experimental feature that is still under development.

4.3.1 Component Bound Branching Rule

When branching on master variables, one needs to define a function $f(j)$, $j \in P \cup R$, that guarantees that the left-hand side of (6) is fractional whenever the solution to the restricted master problem is fractional. Typically, $f(j)$ just identifies a subset of master variables. The *component bound branching* rule [106] is a generic example for such a function. In essence, the rule branches on sums of master variables λ_j for which the components of the corresponding x_j, $j \in P \cup R$, satisfy given lower and upper bounds, see [32] for details.

This rule is implemented in GCG 4.0 as a code example for how to use the newly introduced extended master constraints. The documentation contains a section on "how to add master-only constraints" with details about the interface methods that need to be implemented.

4.4 JDEC Decomposition File Format

GCG supports various decomposition file formats. Users can provide GCG with their own predefined decompositions or let GCG write detected ones to files. The `DEC` file format is easy to use as `.dec` files are simple text files. However, this makes adding additional and advanced features difficult. GCG 4.0 supports a new JSON-based decomposition file format, called `JDEC` using the file extension `.jdec`. JSON is a straightforward and widely adopted file format that is human-readable. Data is stored using arrays and key-value pairs. Many modern programming languages support reading and writing JSON files either natively or by using an open-source library. Hence, a great advantage is that the basic structure of `.jdec` files can be read in without any specialized parser. Currently, `.jdec` files support the following main features, which we plan to extend in the future:

– partial and complete decompositions,
– declaring constraints as master or block constraints,
– specifying nested structures, i.e., decompositions can be provided for blocks,
– storing of symmetry information.

The root object of a `.jdec` file stores metadata of the decomposition and supports the following data fields:

name	description
`version`	version of the file format
`problem_name`	name of the problem the decomposition belongs to
`description`	description of the decomposition
`decomposition`	the decomposition (see below)
`decomposition_id`	internal ID of the decomposition, only written by GCG

The root object stores the actual (root) decomposition using a decomposition object. Decomposition objects always have the same data fields and can be used to specify decompositions of blocks as well. A decomposition object of a `.jdec` file supports the following data fields:

name	description
`n_blocks`	number of blocks of the decomposition, ignored by GCG
`presolved`	indicates whether the decomposition refers to a presolved problem
`master_constraints`	list of master constraints
`blocks`	list of block objects (see below)
`symmetry_var_mapping`	a mapping that maps all variables to their representatives

A block object of a `.jdec` file is used to store information about a block. As already mentioned, users may provide a decomposition for blocks, leading to nested structures. A block object supports the following data fields:

name	description
`index`	index of the block
`constraints`	list of constraints assigned to the block
`decomposition`	decomposition (object) of the block
`symmetry_representative_block`	index of the representative block this block should be mapped to

When reading a `.jdec` file GCG ignores unknown data fields. Thus, users can add additional information, that they need to work with, to their custom decompositions. Similarly, we plan to extend the corresponding reader plugin such that GCG includes additional information about the detection process in the written file. Listing 2 shows an example `.jdec` file that stores a nested decomposition for the `.lp` file depicted in Listing 1. For more details we refer to GCG's documentation[2].

Listing 1: truncated `.lp` file example

```
Minimize
  obj:  + y#0 + y#1 + y#2 + y#3
Subject to
  assign_0:
   + x#0#0 + x#1#0 + x#2#0 + x#3#0 >= 1
  assign_1:
   + x#0#1 + x#1#1 + x#2#1 + x#3#1 >= 1
  link_0:
   - y#0 + y#1 >= 0
  link_1:
   - y#2 + y#3 >= 0
  cap_0:
   -10 y#0 + 2 x#0#0 + 3 x#0#1 <= 0
  cap_1:
   -5 y#1 + 2 x#1#0 + 3 x#1#1 <= 0
  cap_2:
   -10 y#2 + 2 x#2#0 + 3 x#2#1 <= 0
  cap_3:
   -5 y#3 + 2 x#3#0 + 3 x#3#1 <= 0
```

Listing 2: `.jdec` file example

```
{
  "version": 1,
  "name": "example_nested_dec",
  "problem_name": "example.lp",
  "description": "nested decomposition with aggregated blocks",
  "decomposition": {
    "presolved": false,
    "n_blocks": 2,
    "master_constraints": [
      "assign_0"
    ],
    "blocks": [
      {
        "index": 0,
        "constraints": [
          "cap_0",
          "cap_1",
          "link_0"
        ],
        "decomposition": {
          "presolved": false,
          "n_blocks": 2,
          "master_constraints": [
            "link_0"
          ],
          "blocks": [
            {
              "index": 0,
              "constraints": [
                "cap_0"
              ]
            },
            {
              "index": 1,
              "constraints": [
                "cap_1"
              ]
            }
          ]
        },
        "symmetry_representative_block": 0
      },
      {
        "index": 1,
        "constraints": [
          "cap_2",
          "cap_3",
          "link_1"
```

[2]gcg.or.rwth-aachen.de

```
        ],
        "decomposition": {
          "presolved": false,
          "n_blocks": 2,
          "master_constraints": [
            "link_1"
          ],
          "blocks": [
            {
              "index": 0,
              "constraints": [
                "cap_2"
              ]
            },
            {
              "index": 1,
              "constraints": [
                "cap_3"
              ]
            }
          ]
        },
        "symmetry_representative_block": 0,
      }
    ],
    "symmetry_var_mapping": {
      "y_0": "y_0",
      "x_0_0": "x_0_0",
      "x_0_1": "x_0_1",
      "y_1": "y_0",
      "x_1_0": "x_0_0",
      "x_1_1": "x_0_1",
      "y_2": "y_0",
      "x_2_0": "x_0_0",
      "x_2_1": "x_0_1",
      "y_3": "y_0",
      "x_3_0": "x_0_0",
      "x_3_1": "x_0_1"
    }
  }
}
```

4.5 Pricing Problem Solvers

This release adds two new pricing problem solvers: the GCG pricing problem solver and the HiGHS pricing problem solver. In addition, we revised the Cliquer pricing solver.

4.5.1 GCG Pricing Problem Solver

The GCG pricing problem solver solves general MILP pricing problems using GCG. The common use case is to specify a nested structure using the new `JDEC` file format. Then, if enabled, the GCG pricing solver uses this structure to reformulate and solve the pricing problems. Currently, only Dantzig-Wolfe reformulations are supported. The solver can be enabled by setting the parameter `pricingsolver/gcg/maxdepth`, which refers to the maximal recursion depth, to a value greater than zero. Until this depth is reached, the solver either uses the structure information provided by the current (parent) decomposition or tries to detect a structure. The detection takes only place if no nested structure information is provided for all blocks of the parent decomposition, and the maximum depth is not reached yet.

4.5.2 HiGHS Pricing Problem Solver

This pricing solver uses the HiGHS MILP solver [60] to solve the pricing problems. The use of the HiGHS pricing solver is an optional feature and has to be enabled at compile time. It is necessary to have a separate installation of HiGHS[3], which is open-source. If enabled, it will be used to solve general MILP pricing problems instead of the default MILP pricing problem solver using SCIP.

4.5.3 Cliquer Pricing Problem Solver

The Cliquer pricing problem solver is a specialized heuristic pricing solver. It is applicable to weighted independent set pricing problems and closely related variants. The solver uses the Cliquer library[4] that provides functionality to find maximum weighted cliques. We revised large parts of the implementation and improved variable aggregation, the handling of fixed variables, and the check of pricing problem compatibility. Moreover, we added more parameters that allow to control whether the solver should run or reject the solving request. These parameters are based on properties of the constructed graph, such as number of nodes or density.

4.6 Pricing Parallelization

In the past, GCG's parallelization was disabled by default at compile time. With GCG 4.0, we have revised the implementation of GCG's pricing parallelization. If supported by the build environment, this feature is now enabled by default. The parameter `pricing/masterpricer/nthreads` controls the maximal number of threads GCG can use for parallelization. Setting the parameter to 0 means that GCG is allowed to use all available logical cores. At runtime, GCG informs the user about how many threads are used. Note that using more than one thread may lead to nondeterministic behavior, i.e., the solving path for the same input may differ. Thus, by default, GCG uses only one thread. We intend to add a deterministic mode for pricing parallelization in the future.

4.7 IPColGen Primal Heuristic

IPColGen is a matheuristic proposed by Maher and Rönnberg [75] and is implemented in GCG 4.0. It is based on an LNS heuristic framework and uses an adapted pricing scheme. The employed column generation strategy aims at finding columns that lead to high-quality primal feasible solutions for the original problem. The heuristic is applicable if the master problem is a set covering, set packing, or set partitioning problem. For the implementation of IPColGen, GCG's API and data structures were extended. Since IPColGen adapts the pricing scheme, new callback methods are introduced with GCG 4.0. These allow users to influence GCG's pricing procedure and are called before or after GCG's pricing loop. The callback methods are used in the IPColGen heuristic to inject custom weights used by the subsequent pricing procedure. For more details of the IPColGen heuristic we refer to [75]. The heuristic is currently disabled by default, but can be enabled by changing the parameter `heuristics/ipcolgen/freq` in the master problem.

4.8 Decomposition Score Plugin

GCG computes scores to compare decompositions. Based on the resulting ranking a decomposition is picked for the reformulation of the original problem. The implementation of the scores has been refactored[5]. Scores are now implemented as plugins using the data

[3] `https://github.com/ERGO-Code/HiGHS`
[4] `https://users.aalto.fi/~pat/cliquer.html`
[5] this happened already in GCG version 3.6, but was not documented

structure `GCG_SCORE` and can be included by calling `GCGincludeScore`. This simplifies the addition and handling of scores. When working with the CLI, users can display implemented scores with the `display scores` command. The currently selected score can be changed by setting the parameter `detection/scores/selected`.

5 SCIP-SDP

SCIP-SDP is a solver for handling mixed-integer semidefinite programs (MISDPs), without loss of generality, written in the following form

$$\begin{aligned} \inf \quad & b^\top y \\ \text{s.t.} \quad & \sum_{k=1}^{m} A^k\, y_k - A^0 \succeq 0, \\ & \ell_i \le y_i \le u_i && \text{for all } i \in [m], \\ & y_i \in \mathbb{Z} && \text{for all } i \in \mathcal{I}, \end{aligned} \tag{7}$$

with symmetric matrices $A^k \in \mathbb{R}^{n\times n}$ for $i \in \{0,\dots,m\}$, $b \in \mathbb{R}^m$, $\ell_i \in \mathbb{R} \cup \{-\infty\}$, $u_i \in \mathbb{R} \cup \{\infty\}$ for all $i \in [m] \coloneqq \{1,\dots,m\}$. The set of indices of integer variables is given by $\mathcal{I} \subseteq [m]$, and $M \succeq 0$ denotes that a matrix M is positive semidefinite. The development of SCIP-SDP has been described in the earlier SCIP reports.

The current version of SCIP-SDP is 4.4.0, which incorporates updates for SCIP 10. Moreover, it features the possibility to write the original and transformed problem in the *Conic Benchmark Format* (CBF), which has been developed in conjunction with the Conic Benchmark Library, see https://cblib.zib.de/. In this way, also presolved problems can be written.

In addition, we mention the Python code of Johanna Skåntorp at https://github.com/J-Skantorp/CBF-pythonic, which allows to formulate MISDPs and write them in CBF, which then can be read by SCIP-SDP.

6 PaPILO

The solver-independent presolving library PaPILO [47] for MIP and LP has been shipped with the SCIP Optimization Suite since SCIP 7.0. It hooks into the solving process of SCIP through the presolver plugin `milp`. As the only presolving method in SCIP, it also supports multi-precision and rational arithmetic and is, therefore, an important plugin when SCIP is run in exact solving mode, see Section 3. However, as a solver-independent library, PaPILO is also used outside of the SCIP Optimization Suite. Examples are its use as a presolver for the first-order linear programming method PDLP [12], and notably for presolving in the PB solver MIXED-BAG, which won the opt-lin category in the Pseudo-Boolean Competition 24 [94]. In the same competition, PaPILO also participated as part of the SCIP entry, reaching first or second places in linear and nonlinear categories.

The latest version PaPILO 3.0 comes with significant improvements in performance and numerical stability for floating-point arithmetic. In the following, we describe the improvements of performance and memory consumption of the *dominated columns* presolver and introduce the newly included presolver *clique merging*.

The presolvers in PaPILO are parallelized using a transaction-based design [47], in which changes are first recorded as so-called transactions and applied later in a sequential synchronization phase. This introduces challenges, requiring developers to ensure that transactions

1. are stored in a reasonable order (effectiveness) and
2. are generated without excessive redundancy (efficiency),

especially to avoid conflicts among reductions of the same presolving module. Redundant transactions may be generated, for example, by the dominated columns presolver [6] due to the transitivity of dominations. This means, dominations $x_1 \xrightarrow{\text{dom}} x_2$ and $x_2 \xrightarrow{\text{dom}} x_3$ imply the domination $x_1 \xrightarrow{\text{dom}} x_3$ but not all these dominations are required to fix the dominating columns x_2 and x_3. Previous versions of PaPILO store all of the detected dominations without regarding potential redundancies. This can cause memory issues and substantial conflicts especially on feasibility problems, on which usually more redundant dominations exist than on problems with a nonzero objective. In PaPILO 3.0, we apply a *topological compression* to the set of column domination arcs in order to avoid storing redundant dominations as explained next. This keeps the required memory linear in the problem size and substantially reduces the computation time, while maintaining the same number of applied reductions in practice.

One requirement of the transaction-based design is that the bounds of a dominated column must be locked so that applying the domination by fixing the dominating column remains a valid reduction. Due to the lock, a dominating column can only be fixed once. Therefore, we aim at a directed forest on the column set of applicable dominations because in this structure no conflicts can arise. To keep the detection of dominations parallelized and deterministic, the detected dominations are locally collected in thread-separate buckets. The detection is continued until the number of collected dominations reaches the number of columns because then there exists a redundant domination as explained above. Subsequently, this set of dominations is compressed by adding the dominations one by one to a directed forest structure while filtering out dominations that would lead to a directed cycle. The detection and compression is repeated until all possible dominated columns are considered. Afterwards, the resulting set of dominations is converted into a transaction sequence without conflicts by iterating through the domination trees in topological order by a breadth-first search.

The following examples highlight the resulting performance improvements:

– On the instance neos-94313 [46] the number of transactions decreases from 223 200, of which only 6.2 % are applied, to now 13 950 transactions, which are all applied.
– On the instance normalized-aries-da_network_1000_5__369_766__256 [93] the number of transactions decreases from 163 200 000, of which only 0.8 % are applied, to now 1 275 000 transactions, which are all applied.
– On the instance series normalized-PB09/OPT-LIN/aries-da_nrp/normalized-aries-da_network_1000_5__369_766 [93] for indices 512, 1024, and 2048 with up to 10 240 998 variables, presolver DominatedColumns in PaPILO exceeded a memory limit of 48 GB while SCIP with DominatedColumns in PaPILO disabled was not able to solve them within reasonable time. With these changes even such large-scale instances are solved by default to optimality within the given memory limit.

In all cases, the number of transactions is significantly reduced, and the resulting transactions can be applied without conflicts, thereby drastically reducing memory demand.

The new clique merging presolver is implemented similarly to the approach described by Achterberg in [4]. Cliques are constraints which consist of binary variables where at most one variable can be assigned the value one. In the presolver, we first construct a so-called *clique table* [1], which is a graph structure, where each binary variable is represented by a vertex, and an edge connects each pair of variables that belong to the same clique. Then a greedy maximum clique heuristic expands cliques that are already represented by other cliques. Extending cliques can make some cliques redundant, which are then labeled as such and are eventually removed.

In order to limit the memory consumption, the algorithm is restricted to cliques with a certain size. Since clique merging in PaPILO 3.0 is executed in parallel, it is considerably faster than the pre-existing clique merging presolver in SCIP. The current implementation is limited by the fact that extended cliques are copied back to the data structure of the reduced problem. However, the current data structure in PaPILO does not support large extensions of constraints, leading to the necessity of discarding some reductions. This issue could be addressed in the future by maintaining a separate clique table.

7 SoPlex

SoPlex [110] is a simplex-based open-source LP solver and serves as the default underlying solver for LP relaxations within SCIP. In particular, since SoPlex supports solving LPs exactly over the rational numbers, it plays an important role in the exact solving mode of SCIP, cf. Section 3.1. Although no algorithmic improvements have been added since the release of SCIP Optimization Suite 9.0, a new major version SoPlex 8.0 is now released to signify improvements of the build system and needed adjustments of the public API. For details, see the SoPlex changelog.

8 The UG Framework

This release includes a UG application for solving Pseudo-Boolean problems in parallel with FiberSCIP [84]. A boolean parameter `PBCompetionOutputFormat` was added to adjust the output to that of the 2024 Pseudo-Boolean Competition[6] (default value is "FALSE").

9 ZIMPL

With this release we updated Zimpl to Version 3.7.0. Apart from some small bug fixes, Zimpl can now compute permutations. Given a set `A`, `permutate(A)` generates a set consisting of tuples with all permutations of the elements of `A`. This simplifies modeling for problems where one needs all the permutations, e.g., Birkhoff decompositions. Also, variables declared as implicit integer are now recognized by SCIP and handled appropriately.

10 Interfaces

10.1 PySCIPOpt

The main new feature of PySCIPOpt [76], SCIP's Python interface, is the support of matrix variables. In some optimization subfields and other adjacent fields, it is standard practice to both view and model the optimization problem from a matrix perspective. The previous requirement of using individual variables often created a disconnect for such practitioners between the model and its implementation. With this in mind, the concept of matrix variables has been added to PySCIPOpt. The matrix variable structure is built upon the popular `numpy.ndarray`, and thus NumPy has become a required dependency. Matrix variables act in a similar manner to standard variables, and can be used in an expected fashion to create intermediate expressions involving other variables, as well as

[6] https://www.cril.univ-artois.fr/PB24/

for creating both linear and general nonlinear constraints. The syntax resembles that of single variables, using the methods `addMatrixVar` instead of `addVar` and `addMatrixCons` instead of `addCons`. This allows for an easy transition for users familiar with the existing interface. For tutorials on how to use this functionality, visit the Matrix API tutorial in the documentation.

In addition to matrix variables, work was put into helping new users get started with the software. This was achieved by three main means: improving the documentation, creating exercises, and including recipes. The documentation [103] was reworked to have a more aesthetic style, and to include many new tutorials and examples. In the same vein, scipdex [33] was created, with the goal of offering a more interactive way for users to familiarize themselves with PySCIPOpt. It contains exercises of varying difficulty, facilitating the exploration of different aspects of SCIP.

Further, in an attempt to provide additional functionality to users, the recipes subpackage was created. It consists of common features that newcomers often have trouble implementing. Two such examples are (1) requiring an epigraph reformulation to add a nonlinear objective and (2) plotting the evolution of the primal and dual bounds, for example. These were not added directly to PySCIPOpt's source code as to reduce divergence from SCIP.

10.2 SCIP.jl

SCIP is available in Julia through the SCIP.jl package, which exposes both a low-level interface mirroring exactly the C API and a high-level interface based on MathOptInterface [72]. The high-level interface now exposes the creation of SCIP event handlers, allowing users to customize the execution of the solving process with more flexibility. The high-level interface also includes the possibility to compute a set of Minimum Unsatisfiable Constraints (MinUC), i.e., a set of constraints which cannot be satisfied in the current problem. The SCIP.jl implementation of MinUC computes the set of unsatisfiable constraints and then allows users to query the status of each constraint to determine whether they are in the conflict, see, e.g., [23, 90].

10.3 JSCIPOpt

The Java interface to SCIP, JSCIPOpt [62], has been extended to make the following functionality of the SCIP C API newly available from Java:

- the `SCIPgetStage` getter method (contributed by the GitHub user fuookami (Sakurakouji Sakuya)),
- the getters `SCIPgetBoolParam`, `SCIPgetIntParam`, `SCIPgetLongintParam`, `SCIPgetRealParam`, `SCIPgetCharParam`, `SCIPgetStringParam`, `SCIPgetPrimalbound`, `SCIPgetSolvingTime`, `SCIPgetNOrigVars`, and `SCIPgetOrigVars`,
- algebraic expressions (`SCIPcreateExpr*` and `SCIPreleaseExpr` methods),
- all built-in constraint types (`SCIPcreateConsBasic*` methods), e.g., nonlinear constraints (using the algebraic expressions), SOS constraints, logical constraints, etc.

In addition, experimental JSCIPOpt support for writing custom expression handlers in Java, using the `ObjExprHdlr` interface added to the `objscip` C++ API in SCIP 10.0, is available in a branch.

Please note that support for SCIP versions prior to 8.0, which was already broken in practice, was officially dropped from JSCIPOpt, and the minimum CMake version increased to 3.3, which was the minimum required by SCIP 8.0. This allowed fixing some deprecation warnings.

10.4 russcip

At the time of this release, the Rust interface to SCIP, `russcip`, is at Version 0.8.2 bringing many quality-of-life improvements since the previous release. It introduces the new `bundled` feature, which allows users to access a precompiled version of SCIP and its dependencies on all major platforms. Additionally, the `russcip` library has been updated to provide a more ergonomic API, providing a more expressive way to build models. The new API has default values for variable and constraint data, and only the needed ones are passed. This also ensures correct types are used at compile time, e.g., an integer variable can only be created with integer bounds. Next is an example of the new API in comparison to the old one.

```rust
use russcip::prelude::*;

let mut model = Model::default().minimize();

let x = model.add_var(0.0, 1.0, 1.0, "x", VarType::Binary); // previous
    API
let y = model.add(var().bin().obj(1.0)); // add binary variable with
    objective coefficient 1.0

model.add(cons().coef(&x, 1.0).coef(&y, 1.0).eq(1.0)); // x + y = 1
model.add_cons(vec![&x, &y], &[1.0, 1.0], 1.0, 1.0, "cons"); // previous
    API
```

The safe rust API has also grown significantly, providing access to the separator and constraint handler plugins, and methods to build and add cuts. Access to a `Model` object in the solving stage is now only available through the plugin callbacks (trait methods), which enables separation of methods only available in the solving stage to only be called in plugin implementations. Finally, the release brings a simpler memory management model. All SCIP wrapper objects now contain a reference-counted pointer to the SCIP instance they are generated from ensuring that they are valid for the lifetime of the instance.

10.5 Further Interfaces

Next to the previously mentioned interfaces, there also exist interfaces of SCIP for Matlab [79], AMPL, and the C++ interface SCIPPP [99]. The LP solver SoPlex features the Python interface PySoPlex [92]. The presolver PaPILO now also has a file-based interface in Julia for presolving and postsolving instances [87].

11 MIP-DD

MIP-DD is a C++-package that helps to simplify debugging optimization software by reducing the instance triggering a bug. MIP-DD represents the first open-source and solver-independent *delta debugger* for mixed-integer programming solvers [55].

Debugging optimization solvers, such as SCIP, can be quite challenging with many modules interacting in a complex way. A general technique to track down bugs in any kind of software is the implementation of assertions, which are boolean expressions expected to be true signaling an error if realized to be false. Another approach specific to optimization software is the utilization of a so-called debug solution mechanism, which for a given feasible solution signals an error if the solver withdraws it as optimal solution candidate. Depending on how familiar a developer is with the relevant code, identifying the reason of an error on large instances can be difficult and time-consuming. Furthermore, the above debugging methods can only be applied if the failing instance is available. Due to legal reasons or concerns about sensitive information, users may be unable to share an instance with the development team. Therefore, a benefit of an automatic simplification method is that it can destroy relevant sensitive information if it is irrelevant to reproduce an issue, allowing external developers to investigate the issue. Another advantage of simplified instances is that they can be simple enough to be included in regular test sets to ensure that an error is not reintroduced later accidentally. As an example, the instances generated for SCIP can be found in its repository at `check/instances/Issue`, which was added to the regular ctest pipeline.

Delta debugging [111] follows a hypothesis-trial-result approach to isolate the cause of a solver failure by simplifying the input data. This approach has been successfully applied to facilitate debugging SAT and SMT solvers [19, 20, 67, 86, 89].

The primary goal of MIP-DD is to simplify debugging. The main approach is to iteratively simplify the problem instance while ensuring that the error remains reproducible. This can lead to substantially shorter debug logs and also simpler numerics, which results in a streamlined debugging process. As demonstrated in the case studies in Hoen et al. [55], instances triggering fundamental bugs in SCIP can often be reduced by MIP-DD within a reasonable time to an instance comprising just a few variables and constraints.

The latest version MIP-DD 2.0 features an automatic adaption of modification batches, which aims at keeping the total solving effort spent in a modifier run constant over the MIP-DD run based on the solving effort of the current simplified instance. Additionally, it features an automatic restriction of solving limits to terminate solving earlier if an issue should have occurred already based on the previous solving run. Especially these features can significantly accelerate the delta debugging process. Furthermore, it is shipped with an interface to SoPlex as well as SCIP, both for their real and exact solving modes, while there is a separate compilation parameter to select the arithmetic type of the internal data representation in order to improve reproducibility outside MIP-DD. Moreover, there is an option to use the implemented delta debugging algorithm to heuristically compute an IIS, similar to the functionality available in SCIP (see Section 3.9).

In the SCIP releases 8.0.1 to 8.1.1 MIP-DD significantly supported fixing 24 out of all 51 MIP-related bugs documented in the changelog [55]. Up to now, MIP-DD has evolved into a highly useful debugging tool that is actively used in current SCIP development.

MIP-DD is publicly available at `https://github.com/scipopt/MIP-DD`. For a more detailed description of the basic features of MIP-DD, please refer [55].

12 Applications

SCIP 10.0 contains a new application that provides tailored functionality for Pseudo-Boolean optimization problems: the PBSolver. In the following, we provide the main features of this application.

To participate in the Pseudo-Boolean competition, a solver is required to strictly comply with the given regulations as prescribed by the organizers [94]. Especially, the solver has to produce a DIMACS CNF styled log output to correctly communicate with the competition environment. For this reason, an application called PBSolver is created to make this interface public and maintained for simpler future submissions. This application mainly provides the following basic features:

– message handler to prepend each standard log line by the comment specifier “c” in order to retrieve relevant solving statistics for posterior investigations

– event handler to catch events of type SCIP_EVENTTYPE_BESTSOLFOUND in order to directly signal a computation of a better primal solution by the solution specifier "o" followed by its precise objective value

Fixed-size arithmetic can only handle numbers up to a certain magnitude without loss of integral precision. Therefore, the OPB reader was extended by the parameter `reading/opbreader/maxintsize`. If an intsize is given in the instance header which exceeds the parameter value, the status specifier "s UNSUPPORTED" is displayed and the run is immediately terminated. With this application both OPB and WBO instances can be solved out of the box in accordance to competition rules. In the Pseudo-Boolean competition 2024, solvers incorporating SCIP won the categories OPT-LIN, PARTIAL-LIN, SOFT-LIN, OPT-NLC, and DEC-NLC. Out of a total of 1,207 instances, SCIP successfully solved 759, while its parallelized variant FiberSCIP even solved 776. Details on the algorithms useful for Pseudo-Boolean solving which are now implemented in SCIP can be found in [84]. Yet missing but planned to be added in the near future are a problem type detection to apply specific solving parameter settings, a dedicated feasibility definition that allows to solve also numerically challenging instances reliably, and an exact solving mode to also participate in verified categories next time.

Acknowledgments

The authors want to thank all previous developers and contributors to the SCIP Optimization Suite and all users that reported bugs and often also helped reproducing and fixing the bugs. Ambros Gleixner and Leon Eifler wish to thank Kati Jarck and Daniel Steffy for their prototype of the exact solving mode described in [25, 61], in particular their implementation of safe dual bounding described in Section 3.1.4. We like to thank Zuse Institute Berlin and RWTH Aachen for the computing infrastructure that was used for countless test runs and hosting the projects of the suite.

Contributions of the Authors

The material presented in the article is highly related to code and software. In the following we try to make the corresponding contributions of the authors and possible contact points more transparent.

Mathieu Besançon implemented parts of the Probabilistic lookahead strong branching (see Section 3.7) and maintains the Julia interface. Ksenia Bestuzheva implemented the interface to the CONOPT nonlinear programming solver (see Section 3.10). Sander Borst and Leon Eifler contributed safe dual proof analysis and constraint propagation (see Section 3.1.6). Antonia Chmiela provided the first version of the repair mechanism described in Section 3.1.7. João Dionísio has been developing and maintaining PySCIPOpt. Johannes Ehls revised the implementation of GCG's Cliquer pricing problem solver. Leon Eifler was the main author of SCIP's new exact solving mode, with contributions and revisions by Ambros Gleixner and Dominik Kamp, and a concerted code review effort of the whole team (see Section 3.1). Mohammed Ghannam implemented the serialization of SCIP statistics and is one of the developers of PySCIPOpt. Adrian Göß implemented, tested, and finetuned the original heuristic (Section 3.6.1) and its extension (Section 3.6.2). Alexander Hoen, Jacob von Holly-Ponientzietz and Dominik Kamp maintained PaPILO and implemented the features described in Section 6. Alexander Hoen and Dominik Kamp also developed MIP-DD (see Section 11). Christopher Hojny implemented the generalization of symmetry handling methods to reflection symmetries as well as symmetry detection callbacks for Pseudo-Boolean and disjunctive constraints. Rolf van der Hulst revised detection and handling of implied integrality in SCIP (see Section 3.2). Dominik Kamp provided numerous bugfixes all across SCIP, SoPlex, and PaPILO. Thorsten Koch added the permutation feature to ZIMPL (see Section 9). Kevin Kofler maintains the Java interface JSCIPOpt. Jurgen

Lentz developed a draft of the GCG structure and implemented the decomposition score plugin of GCG. He is also a maintainer of PyCGCOpt. Marco Lübbecke coordinated and supervised the development of GCG. Stephen J. Maher implemented the updates to the Benders' decomposition framework (see Section 3.8), the HiGHS pricing solver (Section 4.5.2), and the IPColGen heuristic in GCG (Section 4.7). Paul Matti Meinhold implemented the dynamic batch sizing for the Greedy IIS finder of SCIP. Gioni Mexi implemented the cut-based conflict analysis (see Section 3.4), the probabilistic lookahead strong branching (see Section 3.7), and together with Dominik Kamp and Alexander Hoen the Pseudo-Boolean application (see Section 12). Til Mohr implemented GCG's extended master constraints interface and added the component bound branching rule to GCG. Erik Mühmer implemented the GCG pricing problem solver, GCG's new `JDEC` decomposition file format, and the GCG structure. Furthermore, he revised the pricing parallelization and the extended master constraints interface. Krunal Kishor Patel implemented the Ancestral pseudocosts branching rule. Marc E. Pfetsch maintains SCIP-SDP, wrote the interface of SCIP to DEJAVU, and contributed at several places, including the symmetry code and diverse bug-fixes. Sebastian Pokutta coordinated and supervised multiple developers at ZIB. Chantal Reinartz Groba adapted GCG's extended master constraints interface and added support for separators that work only on the master problem. Felipe Serrano contributed to the development of the cut-based conflict analysis (see Section 3.4). Yuji Shinano worked on the application for Pseudo-Boolean solving in UG (see Section 8). Mark Turner implemented the IIS finder plugin and was one of the maintainers of PySCIPOpt. Stefan Vigerske implemented mixing branching on integer and nonlinear variables, the bound check for aggregated variables, and writing of `.nl` files. Matthias Walter implemented the new multilinear separator (see Section 3.5). Dieter Weninger and Adrian Göß developed the ideas of the Decomposition Kernel Search heuristic (Section 3.6.2). Liding Xu extended the LPI interface of PySCIPOpt.

References

[1] T. Achterberg. *Constraint Integer Programming*. Dissertation, Technische Universität Berlin, 2007.

[2] T. Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 4 (1):4–20, 2007.

[3] T. Achterberg. SCIP: Solving Constraint Integer Programs. *Mathematical Programming Computation*, 1(1):1–41, 2009. doi:10.1007/s12532-008-0001-1.

[4] T. Achterberg. Combinatorial algorithms used inside a MIP solver, 2022. URL `https://sea2022.ifi.uni-heidelberg.de/talk_achterberg.pdf`.

[5] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, 2005.

[6] T. Achterberg, R. E. Bixby, Z. Gu, E. Rothberg, and D. Weninger. Presolve reductions in mixed integer programming. *INFORMS Journal on Computing*, 32(2):473–506, 2020. doi:10.1287/ijoc.2018.0857.

[7] M. Anders. *Efficient Algorithms for Symmetry Detection*. PhD thesis, TU Darmstadt, 2024.

[8] M. Anders and P. Schweitzer. Parallel computation of combinatorial symmetries. In *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 6:1–6:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ESA.2021.6.

[9] M. Anders, P. Schweitzer, and J. Stieß. Engineering a Preprocessor for Symmetry Detection. In L. Georgiadis, editor, *21st International Symposium on Experimental Algorithms (SEA 2023)*, volume 265 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:21, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SEA.2023.1.

[10] E. Angelelli, R. Mansini, and M. G. Speranza. Kernel search: a general heuristic for the multi-dimensional knapsack problem. *Computers & Operations Research*, 37(11): 2017–2026, 2010. doi:10.1016/j.cor.2010.02.002.

[11] E. Angelelli, R. Mansini, and M. G. Speranza. Kernel search: A new heuristic framework for portfolio selection. *Computational Optimization and Applications*, 51(1):345–361, 2012. doi:10.1007/s10589-010-9326-6.

[12] D. Applegate, M. Diaz, O. Hinder, H. Lu, M. Lubin, B. O' Donoghue, and W. Schudy. Practical large-scale linear programming using primal-dual hybrid gradient. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 20243–20257. Curran Associates, Inc., 2021. URL `https://proceedings.neurips.cc/paper_files/paper/2021/file/a8fbbd3b11424ce032ba813493d95ad7-Paper.pdf`.

[13] P. Bendotti, P. Fouilhoux, and C. Rottner. Orbitopal fixing for the full (sub-)orbitope and application to the unit commitment problem. *Mathematical Programming*, 186:337–372, 2021. doi:10.1007/s10107-019-01457-1.

[14] M. Bénichou, J.-M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O. Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1:76–94, 1971.

[15] K. Bestuzheva, M. Besançon, W.-K. Chen, A. Chmiela, T. Donkiewicz, J. van Doornmalen, L. Eifler, O. Gaul, G. Gamrath, A. Gleixner, L. Gottwald, C. Graczyk, K. Halbig, A. Hoen, C. Hojny, R. van der Hulst, T. Koch, M. Lübbecke, S. J. Maher, F. Matter, E. Mühmer, B. Müller, M. E. Pfetsch, D. Rehfeldt, S. Schlein, F. Schlösser, F. Serrano, Y. Shinano, B. Sofranac, M. Turner, S. Vigerske, F. Wegscheider, P. Wellner, D. Weninger, and J. Witzig. The SCIP Optimization Suite 8.0. ZIB-Report 21–41, Zuse Institute Berlin, 2021.

[16] R. E. Bixby and D. K. Wagner. An almost linear-time algorithm for graph realization. *Mathematics of Operation Research*, 13(1), 1988.

[17] BOOSTMP. The Boost Multiprecision Library. `https://github.com/boostorg/multiprecision` (accessed June 21, 2025), 2025.

[18] S. Borst, L. Eifler, and A. Gleixner. Certified constraint propagation and dual proof analysis in a numerically exact MIP solver, 2024. URL `https://arxiv.org/abs/2403.13567`.

[19] R. Brummayer and A. Biere. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, SMT '09, pages 1–5, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1670412.1670413.

[20] R. Brummayer, F. Lonsing, and A. Biere. Automated testing and debugging of SAT and QBF solvers. In O. Strichman and S. Szeider, editors, *Theory and Applications of Satisfiability Testing – SAT 2010*, pages 44–57, Berlin, Heidelberg, 2010. Springer.

[21] D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver. In *Proceedings of the 40th annual Design Automation Conference*, pages 830–835, 2003.

[22] K. Cheung, A. Gleixner, and D. E. Steffy. Verifying Integer Programming Results. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 148–160. Springer, 2017. doi:10.1007/978-3-319-59250-3_13.

[23] J. W. Chinneck. *Feasibility and infeasibility in optimization: algorithms and computational methods*, volume 118 of *International Series in Operations Research and Management Sciences*. Springer, 2008.

[24] W. Cook, S. Dash, R. Fukasawa, and M. Goycoolea. Numerically safe Gomory mixed-integer cuts. *INFORMS Journal on Computing*, 21:641–649, 2009. doi:10.1287/ijoc.1090.0324.

[25] W. Cook, T. Koch, D. E. Steffy, and K. Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5 (3):305–344, 2013. doi:10.1007/s12532-013-0055-6.

[26] Y. Crama and E. Rodríguez-Heck. A class of valid inequalities for multilinear 0–1 optimization problems. *Discrete Optimization*, 25:28–47, 2017. doi:10.1016/j.disopt.2017.02.001.

[27] A. Del Pia and S. Di Gregorio. Chvátal rank in binary polynomial optimization. *INFORMS Journal on Optimization*, pages 315–443, 2021. doi:10.1287/ijoo.2019.0049.

[28] A. Del Pia and A. Khajavirad. A Polyhedral Study of Binary Polynomial Programs. *Mathematics of Operations Research*, 42(2):389–410, 2017. doi:10.1287/moor.2016.0804.

[29] A. Del Pia and A. Khajavirad. The multilinear polytope for acyclic hypergraphs. *SIAM Journal on Optimization*, 28(2):1049–1076, 2018. doi:10.1137/16M1095998.

[30] A. Del Pia and M. Walter. Simple odd β-cycle inequalities for binary polynomial optimization. In K. Aardal and L. Sanità, editors, *Integer Programming and Combinatorial Optimization*, pages 181–194. Springer International Publishing, 2022. doi:10.1007/978-3-031-06901-7_14.

[31] A. Del Pia, A. Khajavirad, and N. V. Sahinidis. On the impact of running intersection inequalities for globally solving polynomial optimization problems. *Mathematical Programming Computation*, 12(2):165–191, 2020. doi:10.1007/s12532-019-00169-z.

[32] J. Desrosiers, M. Lübbecke, G. Desaulniers, and J. B. Gauthier. Branch-and-price. Les Cahiers du GERAD G-2024-36, GERAD, 2024. Forthcoming with Springer.

[33] J. Dionísio and M. Ghannam. scipdex. https://github.com/mmghannam/scipdex (accessed May 2, 2025), 2025.

[34] J. van Doornmalen and C. Hojny. A unified framework for symmetry handling. *Mathematical Programming*, 212:217–271, 2025. doi:10.1007/s10107-024-02102-2.

[35] L. Eifler. *Algorithms and Certificates for Exact Mixed Integer Programming*. Dissertation, Technische Universität Berlin, 2025. URL https://doi.org/10.14279/depositonce-23941.

[36] L. Eifler and A. Gleixner. A computational status update for exact rational mixed integer programming. *Mathematical Programming*, 197:793–812, 2023. doi:10.1007/s10107-021-01749-5.

[37] L. Eifler and A. Gleixner. Safe and verified Gomory mixed-integer cuts in a rational mixed-integer program framework. *SIAM Journal on Optimization*, 34(1):742–763, 2024. doi:10.1137/23M156046X.

[38] J. Elffers and J. Nordström. Divide and conquer: Towards faster pseudo-boolean solving. In *IJCAI*, volume 18, pages 1291–1299, 2018.

[39] D. G. Espinoza. *On Linear Programming, Integer Programming and Cutting Planes*. PhD thesis, Georgia Institute of Technology, 2006.

[40] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2):13–es, June 2007. https://doi.org/10.1145/1236463.1236468.

[41] T. Gally, M. E. Pfetsch, and S. Ulbrich. A framework for solving mixed-integer semidefinite programs. *Optimization Methods and Software*, 33(3):594–632, 2018. doi:10.1080/10556788.2017.1322081.

[42] G. Gamrath and M. E. Lübbecke. Experiments with a generic Dantzig-Wolfe decomposition for integer programs. In P. Festa, editor, *Experimental Algorithms*, volume 6049 of *Lecture Notes in Computer Science*, pages 239–252. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-13193-6_21.

[43] G. Gamrath, D. Anderson, K. Bestuzheva, W.-K. Chen, L. Eifler, M. Gasse, P. Gemander, A. Gleixner, L. Gottwald, K. Halbig, G. Hendel, C. Hojny, T. Koch, P. L. Bodic, S. J. Maher, F. Matter, M. Miltenberger, E. Mühmer, B. Müller, M. E. Pfetsch, F. Schlösser, F. Serrano, Y. Shinano, C. Tawfik, S. Vigerske, F. Wegscheider, D. Weninger, and J. Witzig. The SCIP Optimization Suite 7.0. Technical report, Optimization Online, 2020. http://www.optimization-online.org/DB_HTML/2020/03/7705.html.

[44] P. E. Gill, N. I. Gould, W. Murray, M. A. Saunders, and M. H. Wright. A weighted Gram-Schmidt method for convex quadratic programming. *Mathematical programming*, 30(2):176–195, 1984.

[45] W. Glankwamdee and J. Linderoth. Lookahead branching for mixed integer programming. In *Twelfth INFORMS Computing Society Meeting*, pages 130–150, 2006.

[46] A. Gleixner, G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P. M. Christophel, K. Jarck, T. Koch, J. Linderoth, M. Lübbecke, H. D. Mittelmann, D. Ozyurt, T. K. Ralphs, D. Salvagnin, and Y. Shinano. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. *Mathematical Programming Computation*, 2021. doi:10.1007/s12532-020-00194-3.

[47] A. Gleixner, L. Gottwald, and A. Hoen. PaPILO: A parallel presolving library for integer and linear optimization with multiprecision support. *INFORMS Journal on Computing*, 35(6):1329–1341, Nov. 2023. doi:10.1287/ijoc.2022.0171.

[48] GMP. GMP: The GNU multiple precision arithmetic library. `https://gmplib.org` (accessed June 21, 2025), 2025.

[49] D. Goldfarb and A. Idnani. A numerically stable dual method for solving strictly convex quadratic programs. *Mathematical programming*, 27(1):1–33, 1983.

[50] R. E. Gomory. An algorithm for the mixed integer problem. Technical report, RAND Corporation, 1960.

[51] G. Guastaroba, M. Savelsbergh, and M. G. Speranza. Adaptive kernel search: a heuristic for solving mixed integer linear programs. *European Journal of Operational Research*, 263 (3):789–804, 2017. doi:10.1016/j.ejor.2017.06.005.

[52] O. Guieu and J. W. Chinneck. Analyzing infeasible mixed-integer and integer linear programs. *INFORMS Journal on Computing*, 11(1):63–77, 1999.

[53] K. Halbig, A. Göß, and D. Weninger. Exploiting user-supplied decompositions inside heuristics. *Journal of Heuristics*, 31(36), 2025. doi:10.1007/s10732-025-09572-3.

[54] A. Hoen, A. Oertel, A. Gleixner, and J. Nordström. Certifying MIP-based presolve reductions for 0-1 integer linear programs. In B. Dilkina, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 310–328, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-60597-0_20.

[55] A. Hoen, D. Kamp, and A. Gleixner. MIP-DD: Delta debugging for mixed-integer programming solvers. *INFORMS Journal on Computing*, 2025. doi:10.1287/ijoc.2024.0844.

[56] A. J. Hoffman and J. B. Kruskal. Integral Boundary Points of Convex Polyhedra. In *Linear Inequalities and Related Systems. (AM-38)*, volume 38, pages 223–246. Princeton University Press, dec 1957. doi:10.1515/9781400881987-014.

[57] C. Hojny. Packing, partitioning, and covering symresacks. *Discrete Applied Mathematics*, 283:689–717, 2020. doi:10.1016/j.dam.2020.03.002.

[58] C. Hojny. Detecting and handling reflection symmetries in mixed-integer (nonlinear) programming and beyond. *Mathematical Programming Compution*, 2025. doi:10.1007/s12532-025-00289-9.

[59] C. Hojny and M. E. Pfetsch. Polytopes associated with symmetry handling. *Mathematical Programming*, 175(1):197–240, 2019. doi:10.1007/s10107-018-1239-7.

[60] Q. Huangfu and J. Hall. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10:119–142, 2015. doi:10.1007/s12532-017-0130-5.

[61] K. Jarck. *Exact mixed-integer programming.* PhD thesis, TU Berlin, 2020.

[62] JSCIPOpt. JSCIPOpt: The Java interface to SCIP. `https://github.com/scipopt/JSCIPOpt`, 2023.

[63] T. Junttila and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In D. Applegate, G. S. Brodal, D. Panario, and R. Sedgewick, editors, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics*, pages 135–149. SIAM, 2007. doi:10.1137/1.9781611972870.13.

[64] T. Junttila and P. Kaski. Conflict propagation and component recursion for canonical labeling. In A. Marchetti-Spaccamela and M. Segal, editors, *Theory and Practice of Algorithms in (Computer) Systems – First International ICST Conference, TAPAS 2011, Rome, Italy, April 18–20, 2011. Proceedings*, volume 6595 of *Lecture Notes in Computer Science*, pages 151–162. Springer, 2011. doi:10.1007/978-3-642-19754-3_16.

[65] V. Kaibel and M. E. Pfetsch. Packing and partitioning orbitopes. *Mathematical Programming*, 114(1):1–36, 2008. doi:10.1007/s10107-006-0081-5.

[66] V. Kaibel, M. Peinhardt, and M. E. Pfetsch. Orbitopal fixing. *Discrete Optimization*, 8 (4):595–610, 2011. doi:10.1016/j.disopt.2011.07.001.

[67] D. Kaufmann and A. Biere. Fuzzing and delta debugging and-inverter graph verification tools. In L. Kovács and K. Meinke, editors, *Tests and Proofs - 16th International Conference, TAP 2022, Held as Part of STAF 2022, Nantes, France, July 5, 2022, Proceedings*, volume 13361 of *Lecture Notes in Computer Science*, pages 69–88. Springer, 2022. doi:10.1007/978-3-031-09827-7_5.

[68] A. Khajavirad. The circle packing problem: A theoretical comparison of various convexification techniques. *Operations Research Letters*, 57, 2024. doi:10.1016/j.orl.2024.107197.

[69] Y. Kiam and M. Myreen. Formalisation of VIPR in CakeML. `https://github.com/CakeML/cakeml/tree/master/examples/vipr` (accessed May 1, 2024), 2024.

[70] T. Koch. *Rapid Mathematical Prototyping*. Dissertation, Technische Universität Berlin, 2004.

[71] D. Le Berre and A. Parrain. The Sat4j library, release 2.2: System description. *Journal on Satisfiability, Boolean Modelling and Computation*, 7(2-3):59–64, 2011.

[72] B. Legat, O. Dowson, J. D. Garcia, and M. Lubin. MathOptInterface: a data structure for mathematical optimization problems. *INFORMS Journal on Computing*, 34(2):672–689, 2022.

[73] L. Liberti. Reformulations in mathematical programming: automatic symmetry detection and exploitation. *Mathematical Programming*, 131(1–2):273–304, 2012. doi:10.1007/s10107-010-0351-0.

[74] L. Liberti and J. Ostrowski. Stabilizer-based symmetry breaking constraints for mathematical programs. *Journal of Global Optimization*, 60:183–194, 2014.

[75] S. Maher and E. Rönnberg. Integer programming column generation: accelerating branch-and-price using a novel pricing scheme for finding high-quality solutions in set covering, packing, and partitioning problems. *Mathematical Programming Computation*, 15:509–548, 2023. doi:10.1007/s12532-023-00240-w.

[76] S. Maher, M. Miltenberger, J. P. Pedroso, D. Rehfeldt, R. Schwarz, and F. Serrano. PySCIPOpt: Mathematical programming in Python with the SCIP Optimization Suite. In G.-M. Greuel, T. Koch, P. Paule, and A. Sommese, editors, *Mathematical Software – ICMS 2016*, pages 301–307, Cham, 2016. Springer International Publishing.

[77] S. J. Maher. Implementing the branch-and-cut approach for a general purpose Benders' decomposition framework. *European Journal of Operational Research*, 290(2):479–498, 2021.

[78] H. Marchand and L. Wolsey. Aggregation and mixed integer rounding to solve MIPs. *Operations Research*, 49:325–468, 2001. doi:10.1287/opre.49.3.363.11211.

[79] MatlabSCIPInterface. The Matlab interface to SCIP and SCIP-SDP. `https://github.com/scipopt/MatlabSCIPInterface`, 2023.

[80] B. D. McKay and A. Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60:94–112, 2014. doi:10.1016/j.jsc.2013.09.003.

[81] G. Mexi, T. Berthold, A. Gleixner, and J. Nordström. Improving Conflict Analysis in MIP Solvers by Pseudo-Boolean Reasoning. In *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming (CP'23)*, volume 280, page 27, 2023.

[82] G. Mexi, F. Serrano, T. Berthold, A. Gleixner, and J. Nordström. Cut-based conflict analysis in mixed integer programming. *arXiv preprint arXiv:2410.15110*, 2024.

[83] G. Mexi, S. Shamsi, M. Besançon, and P. L. Bodic. Probabilistic lookahead strong branching via a stochastic abstract branching model. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 2024.

[84] G. Mexi, D. Kamp, Y. Shinano, S. Pu, A. Hoen, K. Bestuzheva, C. Hojny, M. Walter, M. E. Pfetsch, S. Pokutta, and T. Koch. State-of-the-art methods for pseudo-boolean solving with SCIP. `https://doi.org/10.48550/arXiv.2501.03390`, 2025.

[85] MPFR. The GNU MPFR library. `https://www.mpfr.org/` (accessed June 21, 2025), 2025.

[86] A. Niemetz and A. Biere. ddSMT: A Delta Debugger for the SMT-LIB v2 Format. In R. Bruttomesso and A. Griggio, editors, *Proceedings of the 11th International Workshop on Satisfiability Modulo Theories, SMT 2013), affiliated with the 16th International Conference on Theory and Applications of Satisfiability Testing, SAT 2013, Helsinki, Finland, July 8-9, 2013*, pages 36–45, 2013.

[87] PaPILO.jl. The Julia interface for PaPILO. `https://www.github.com/scipopt/PaPILO.jl`, 2025.

[88] A. L. Parker and R. R. Hughes. Approximation programming of chemical processes—2: computational difficulties. *Computers & Chemical Engineering*, 5(3):135–141, 1981.

[89] T. Paxian and A. Biere. Uncovering and classifying bugs in MaxSAT solvers through fuzzing and delta debugging. In M. Järvisalo and D. Le Berre, editors, *Proceedings of the 14th Internantional Workshop on Pragmatics of SAT Co-located with the 26th International Conference on Theory and Applicationas of Satisfiability Testing (SAT 2003), Alghero, Italy, July, 4, 2023*, volume 3545 of *CEUR Workshop Proceedings*, pages 59–71. CEUR-WS.org, 2023. URL http://ceur-ws.org/Vol-3545/paper5.pdf.

[90] M. E. Pfetsch. Branch-and-cut for the maximum feasible subsystem problem. *SIAM J. Optim.*, 19(1):21–38, 2008. doi:10.1137/050645828.

[91] Y. Puranik and N. V. Sahinidis. Deletion presolve for accelerating infeasibility diagnosis in optimization models. *INFORMS Journal on Computing*, 29(4):754–766, 2017.

[92] PySoPlex. The Python interface for SoPlex. https://www.github.com/scipopt/PySoPlex, 2023.

[93] O. Roussel. Pseudo-Boolean competition 2009, 2010. URL http://www.cril.univ-artois.fr/PB09/.

[94] O. Roussel. Pseudo-Boolean competition 2024, 2024. URL http://www.cril.univ-artois.fr/PB24/.

[95] D. Ryan and B. Foster. An integer programming approach to scheduling. In A. Wren, editor, *Computer Scheduling of Public Transport Urban Passenger Vehicle and Crew Scheduling*, pages 269–280, Amsterdam, 1981. North-Holland.

[96] D. Salvagnin. A dominance procedure for integer programming. *Master's thesis, University of Padova, Padova, Italy*, 2005.

[97] D. Salvagnin. Symmetry breaking inequalities from the Schreier-Sims table. In W.-J. van Hoeve, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 521–529, Cham, 2018. Springer International Publishing.

[98] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[99] SCIP++. The C++ interface for SCIP. https://www.github.com/scipopt/SCIPpp, 2023.

[100] Y. Shinano. The Ubiquity Generator framework: 7 years of progress in parallelizing branch-and-bound. In N. Kliewer, J. F. Ehmke, and R. Borndörfer, editors, *Operations Research Proceedings 2017*, pages 143–149. Springer, 2018. doi:10.1007/978-3-319-89920-6_20.

[101] P. G. Szabó, M. C. Markót, and T. Csendes. *Global Optimization in Geometry — Circle Packing into the Square*, pages 233–265. Springer US, Boston, MA, 2005. doi:10.1007/0-387-25570-2_9.

[102] K. Truemper. A decomposition theory for matroids. V. Testing of matrix total unimodularity. *Journal of Combinatorial Theory, Series B*, 49(2):241–281, 1990. doi:10.1016/0095-8956(90)90030-4.

[103] M. Turner. PySCIPOpt Documentation. https://pyscipopt.readthedocs.io/en/latest/index.html (accessed May 2, 2025), 2025.

[104] R. van der Hulst and M. Walter. A row-wise algorithm for graph realization. https://optimization-online.org/?p=27423, 2024.

[105] R. van der Hulst and M. Walter. Implied integrality in mixed-integer optimization. https://arxiv.org/abs/2504.07209, To appear in "Integer Programming and Combinatorial Optimization 2025", 2025.

[106] F. Vanderbeck. On Dantzig-Wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm. *Operations Research*, 48(1):111–128, 2000.

[107] VIPR. VIPR: Verifying Integer Programming Results. https://github.com/scipopt/vipr (accessed June 21, 2025), 2025.

[108] M. Walter and K. Truemper. Implementation of a unimodularity test. *Mathematical Programming Computation*, 5(1):57–73, 2013. doi:10.1007/s12532-012-0048-x.

[109] J. Witzig, T. Berthold, and S. Heinz. Computational aspects of infeasibility analysis in mixed integer programming. *Mathematical Programming Computation*, Mar. 2021. doi:10.1007/s12532-021-00202-0.

[110] R. Wunderling. *Paralleler und objektorientierter Simplex-Algorithmus.* Dissertation, Technische Universität Berlin, 1996.

[111] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In O. Nierstrasz and M. Lemoine, editors, *Software Engineering — ESEC/FSE '99*, pages 253–267, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48166-9.

Author Affiliations

Christopher Hojny
Technische Universiteit Eindhoven, Department of Mathematics and Computer Science, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
E-mail: c.hojny@tue.nl
ORCID: 0000-0002-5324-8996

Mathieu Besançon
Université Grenoble Alpes, Inria, Laboratoire d'Informatique de Grenoble, 38100 Grenoble, France
E-mail: mathieu.besancon@inria.fr
ORCID: 0000-0002-6284-3033

Ksenia Bestuzheva
GAMS Software GmbH
E-mail: kbestuzheva@gams.com
ORCID: 0000-0002-7018-7099

Sander Borst
Max Planck Institute for Informatics, Saarland Informatics Campus, Campus E1 4, 66123 Saarbrücken, Germany
E-mail: sborst@mpi-inf.mpg.de
ORCID: 0000-0003-4001-6675

Antonia Chmiela
Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany
E-mail: chmiela@zib.de
ORCID: 0000-0002-4809-2958

João Dionísio
University of Porto, Faculty of Sciences, Rua do Campo Alegre, 4169-007 Porto, Portugal
and Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
E-mail: joao.goncalves.dionisio@gmail.com
ORCID: 0009-0005-5160-0203

Johannes Ehls
RWTH Aachen University, Lehrstuhl für Operations Research, Kackertstr. 7, 52072 Aachen, Germany
E-mail: johannes.ehls@rwth-aachen.de
ORCID: 0009-0005-1130-6683

Leon Eifler
Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany
E-mail: eifler@zib.de
ORCID: 0000-0003-0245-9344

Mohammed Ghannam
Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
E-mail: ghannam@zib.de

Ambros Gleixner
Hochschule für Technik und Wirtschaft Berlin, 10313 Berlin, Germany
and Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
E-mail: gleixner@htw-berlin.de
ORCID: 0000-0003-0391-5903

Adrian Göß
University of Technology Nuremberg, Dr.-Luise-Herzberg-Str. 4, 90461 Nuremberg, Germany
E-mail: adrian.goess@utn.de
ORCID: 0009-0002-7144-8657

Alexander Hoen
Hochschule für Technik und Wirtschaft Berlin, 10313 Berlin, Germany
and Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
E-mail: hoen@zib.de
ORCID: 0000-0003-1065-1651

Jacob von Holly-Ponientzietz
Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
E-mail: von.holly-ponientzietz@zib.de
ORCID: 0009-0002-2601-3689

Rolf van der Hulst
University of Twente, Department of Applied Mathematics, P.O. Box 217, 7500 AE Enschede, The Netherlands
E-mail: r.p.vanderhulst@utwente.nl
ORCID: 0000-0002-5941-3016

Dominik Kamp
University of Bayreuth, Chair of Economathematics, Universitaetsstrasse 30, 95440 Bayreuth, Germany
E-mail: dominik.kamp@uni-bayreuth.de
ORCID: 0009-0005-5577-9992

Thorsten Koch
Technische Universität Berlin, Chair of Software and Algorithms for Discrete Optimization, Straße des 17. Juni 135, 10623 Berlin, Germany, and
Zuse Institute Berlin, Department A^{2}IM, Takustr. 7, 14195 Berlin, Germany
E-mail: koch@zib.de
ORCID: 0000-0002-1967-0077

Kevin Kofler
DAGOPT Optimization Technologies GmbH
E-mail: kofler@dagopt.com

Jurgen Lentz
RWTH Aachen University, Lehrstuhl für Operations Research, Kackertstr. 7, 52072 Aachen, Germany
E-mail: jurgen.lentz@rwth-aachen.de
ORCID: 0009-0000-0531-412X

Marco Lübbecke
RWTH Aachen University, Lehrstuhl für Operations Research, Kackertstr. 7, 52072 Aachen, Germany
E-mail: marco.luebbecke@rwth-aachen.de
ORCID: 0000-0002-2635-0522

Stephen J. Maher
GAMS Software GmbH, Germany

E-mail: smaher@gams.com
ORCID: 0000-0003-3773-6882

Paul Matti Meinhold
Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
E-mail: meinhold@zib.de
ORCID: 0009-0003-5477-9152

Gioni Mexi
Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
E-mail: mexi@zib.de

Til Mohr
RWTH Aachen University, Lehrstuhl für Operations Research, Kackertstr. 7, 52072 Aachen, Germany
E-mail: til.mohr@rwth-aachen.de
ORCID: 0009-0001-9842-210X

Erik Mühmer
RWTH Aachen University, Lehrstuhl für Operations Research, Kackertstr. 7, 52072 Aachen, Germany
E-mail: erik.muehmer@rwth-aachen.de
ORCID: 0000-0003-1114-3800

Krunal Kishor Patel
CERC, Polytechnique Montréal, 2500 Chemin de Polytechnique, Montréal, H3T 1J4, QC, Canada
E-mail: krunal.patel@polymtl.ca
ORCID: 0000-0001-7414-5040

Marc E. Pfetsch
Technische Universität Darmstadt, Fachbereich Mathematik, Dolivostr. 15, 64293 Darmstadt, Germany
E-mail: pfetsch@mathematik.tu-darmstadt.de
ORCID: 0000-0002-0947-7193

Sebastian Pokutta
Technische Universität Berlin, Straße des 17. Juni 135, 10623 Berlin, Germany, and
Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
E-mail: pokutta@zib.de
ORCID: 0000-0001-7365-3000

Chantal Reinartz Groba
RWTH Aachen University, Lehrstuhl für Operations Research, Kackertstr. 7, 52072 Aachen, Germany
E-mail: chantal.michelle.reinartz@rwth-aachen.de
ORCID: 0009-0001-1820-3864

Felipe Serrano
COPT GmbH, Berlin, Germany
E-mail: serrano@copt.de
ORCID: 0000-0002-7892-3951

Yuji Shinano
Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
E-mail: shinano@zib.de
ORCID: 0000-0002-2902-882X

Mark Turner
Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
E-mail: turner@zib.de
ORCID: 0000-0001-7270-1496

Stefan Vigerske
GAMS Software GmbH, c/o Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
E-mail: svigerske@gams.com
ORCID: 0009-0001-2262-0601

Matthias Walter
University of Twente, Department of Applied Mathematics, P.O. Box 217, 7500 AE Enschede, The Netherlands
E-mail: m.walter@utwente.nl
ORCID: 0000-0002-6615-5983

Dieter Weninger
Friedrich-Alexander-Universität Erlangen-Nürnberg, Cauerstr. 11, 91058 Erlangen, Germany
E-mail: dieter.weninger@fau.de
ORCID: 0000-0002-1333-8591

Liding Xu
Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany
E-mail: liding.xu@zib.de
ORCID: 0000-0002-0286-1109