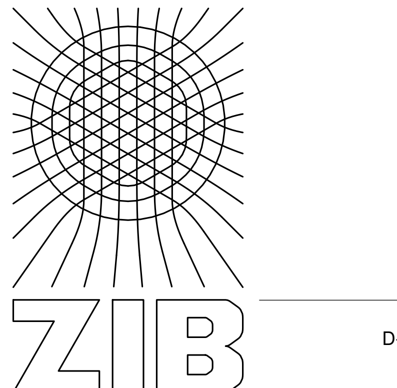


Zuse Institute Berlin

Takustrasse 7
D-14195 Berlin-Dahlem
Germany

GERALD GAMRATH, DANIEL ANDERSON,
KSENIA BESTUZHEVA, WEI-KUN CHEN, LEON EIFLER,
MAXIME GASSE, PATRICK GEMANDER, AMBROS GLEIXNER,
LEONA GOTTWALD, KATRIN HALBIG, GREGOR HENDEL,
CHRISTOPHER HOJNY, THORSTEN KOCH, PIERRE LE BODIC,
STEPHEN J. MAHER, FREDERIC MATTER,
MATTHIAS MILTENBERGER, ERIK MÜHMER,
BENJAMIN MÜLLER, MARC E. PFETSCH,
FRANZISKA SCHLÖSSER, FELIPE SERRANO, YUJI SHINANO,
CHRISTINE TAWFIK, STEFAN VIGERSKE, FABIAN WEGSCHEIDER,
DIETER WENINGER, JAKOB WITZIG

The SCIP Optimization Suite 7.0

ZIB Report 20-10 (March 2020)

Zuse Institute Berlin
Takustrasse 7
D-14195 Berlin-Dahlem

Telefon: 030-84185-0
Telefax: 030-84185-125

e-mail: `bibliothek@zib.de`
URL: `http://www.zib.de`

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

The SCIP Optimization Suite 7.0

Gerald Gamrath · Daniel Anderson · Ksenia Bestuzheva
Wei-Kun Chen · Leon Eifler · Maxime Gasse
Patrick Gemander · Ambros Gleixner · Leona Gottwald
Katrin Halbig · Gregor Hendel · Christopher Hojny
Thorsten Koch · Pierre Le Bodic · Stephen J. Maher
Frederic Matter · Matthias Miltenberger · Erik Mühmer
Benjamin Müller · Marc E. Pfetsch · Franziska Schlösser
Felipe Serrano · Yuji Shinano · Christine Tawfik
Stefan Vigerske · Fabian Wegscheider · Dieter Weninger
Jakob Witzig*

March 30, 2020

Abstract The SCIP Optimization Suite provides a collection of software packages for mathematical optimization centered around the constraint integer programming framework SCIP. This paper discusses enhancements and extensions contained in version 7.0 of the SCIP Optimization Suite. The new version features the parallel presolving library PaPILO as a new addition to the suite. PaPILO 1.0 simplifies mixed-integer linear optimization problems and can be used stand-alone or integrated into SCIP via a presolver plugin. SCIP 7.0 provides additional support for decomposition algorithms. Besides improvements in the Benders' decomposition solver of SCIP, user-defined decomposition structures can be read, which are used by the automated Benders' decomposition solver and two primal heuristics. Additionally, SCIP 7.0 comes with a tree size estimation that is used to predict the completion of the overall solving process and potentially trigger restarts. Moreover, substantial performance improvements of the MIP core were achieved by new developments in presolving, primal heuristics, branching rules, conflict analysis, and symmetry handling. Last, not least, the report presents updates to other components and extensions of the SCIP Optimization Suite, in particular, the LP solver SoPlex and the mixed-integer semidefinite programming solver SCIP-SDP.

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

*Extended author information is available at the end of the paper. The work for this article has been partly conducted within the *Research Campus MODAL* funded by the German Federal Ministry of Education and Research (BMBF grant number 05M14ZAM) and has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 773897. It has also been partly supported by the German Research Foundation (DFG) within the Collaborative Research Center 805, Project A4, and the EXPRESS project of the priority program CoSIP (DFG-SPP 1798). This work was also partly supported by the UK Engineering and Physical Sciences Research Council (EPSRC) grant EP/P003060/1.

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

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 modeling language Zimpl [50],
- the simplex-based linear programming solver SoPlex [79],
- the constraint integer programming solver SCIP [3], which can be used as a fast standalone solver for mixed-integer linear and nonlinear programs and a flexible branch-cut-and-price framework,
- the automatic decomposition solver GCG [29],
- the UG framework for parallelization of branch-and-bound solvers [73], and
- the presolving library PaPILO for linear and mixed-integer linear programs, a new addition in version 7.0 of the SCIP Optimization Suite.

All six tools can be downloaded in source code and are freely available for use in non-profit research. They are accompanied by several extensions for solving specific problem-classes such as the award-winning Steiner tree solver SCIP-Jack [32] and the mixed-integer semidefinite programming solver SCIP-SDP [27]. This paper describes the new features and enhanced algorithmic components contained in version 7.0 of the SCIP Optimization Suite.

Background SCIP has been designed as a branch-cut-and-price framework to solve different types of optimization problems, most importantly, *mixed-integer linear programs* (MIPs) and *mixed-integer nonlinear programs* (MINLPs). MIPs are optimization problems of the form

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

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 $\mathcal{I} \subseteq \mathcal{N} \coloneqq \{1, \ldots, n\}$. The usage of $\overline{\mathbb{R}} \coloneqq \mathbb{R} \cup \{-\infty, \infty\}$ allows for variables that are free or bounded only in one direction (we assume that variables are not fixed to $\pm\infty$).

Another focus of SCIP's research and development are *mixed-integer nonlinear programs* (MINLPs). MINLPs can be written in the form

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

where the functions $f : \mathbb{R}^n \to \mathbb{R}$ and $g_k : \mathbb{R}^n \to \mathbb{R}$, $k \in \mathcal{M} \coloneqq \{1, \ldots, m\}$, are possibly nonconvex. Within SCIP, we assume that f and g_k are specified explicitly in algebraic form using base expressions that are known to SCIP.

SCIP is not restricted to solving MIPs and MINLPs, but is a framework for solving *constraint integer programs* (CIPs), a generalization of the former two problem classes.

The introduction of CIPs was motivated by the modeling flexibility of constraint programming and the algorithmic requirements of integrating it with efficient solution techniques available for MIPs. Later on, this framework allowed for an integration of MINLPs as well. Roughly speaking, 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.

In order to solve CIPs, SCIP relies on the presence of a relaxation—typically the LP relaxation. If the relaxation solution is not feasible for the current subproblem, the enforcement callbacks of the constraint handlers need to take measures to eventually render the relaxation solution infeasible for the updated relaxation, for example by branching or separation. Being a framework for solving CIPs, SCIP can be extended by plugins to be able to solve any CIP. The default plugins included in the SCIP Optimization Suite provide tools to solve MIPs and many MINLPs as well as some classes of instances from constraint programming, satisfiability testing, and pseudo-Boolean optimization. Additionally, SCIP-SDP allows to solve mixed-integer semidefinite programs.

The core of SCIP coordinates a central branch-cut-and-price algorithm. Advanced methods like primal heuristics, branching rules, and cutting plane separators can be integrated as plug-ins with a pre-defined interface. The release version of SCIP comes with many such plug-ins needed to achieve a good MIP and MINLP performance. The solving process is described in more detail by Achterberg [2] and, with focus on the MINLP extensions, by Vigerske and Gleixner [76].

By design, SCIP interacts closely with the other components of the SCIP Optimization Suite. Optimization models formulated in Zimpl can be read by SCIP. PaPILO provides an additional fast and effective presolving procedure that is called from a SCIP presolver plugin. The linear programs (LPs) solved repeatedly during the branch-cut-and-price algorithm are by default optimized with SoPlex. GCG extends SCIP to automatically detect problem structure and generically apply decomposition algorithms based on the Dantzig-Wolfe or the Benders' decomposition scheme. And 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.

New Developments and Structure of the Paper The SCIP Optimization Suite 7.0 introduces the new presolving library PaPILO as its sixth component. It is described in Section 3. Moreover, the other packages of the SCIP Optimization Suite 7.0 provide extended functionality. Updates to SCIP itself are presented in Section 4. The most significant additions and improvements in SCIP 7.0 are

- a tree size estimation method that is used to report an estimate on the amount of the search that has been completed and to trigger restarts,
- a data structure to store decomposition information, used by two primal heuristics and the Benders' decomposition framework,
- the extension of SCIP's Benders' decomposition to handle convex MINLP subproblems, and
- a revision of SCIP's symmetry handling methods that allows the combination of polyhedral methods with orbital fixing.

An overview of the performance improvements for standalone MIP and MINLP is given in Section 2. Section 5 describes the updates in the LP solver SoPlex 5.0. The most recent version 3.0.3 of the generic column generation solver GCG is mainly a bugfix release. Besides smaller fixes, the updates to GCG reflect the changes to the interface of SCIP. The most recent version 3.3.9 of the Zimpl modelling language is a minor bugfix release. The parallelization framework UG 0.8.5 now also provides parallelization for

the Benders' decomposition of SCIP, see Section 6. Release 3.2.0 of SCIP-SDP allows to add the constraint that the matrices are rank-1, see Section 7. Moreover, it enables to upgrade quadratic constraints and features more flexibility with reading and writing files in CBF format.

2 Overall Performance Improvements for MIP and MINLP

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 MIP and MINLP instances is of particular interest during the development process. Additionally, most algorithmic extensions of SCIP like decomposition approaches or problem-specific extensions profit directly from an improved performance on those basic problem classes.

Therefore, computational experiments were performed to evaluate the performance improvement achieved with SCIP 7.0 compared to SCIP 6.0. The methodology and the results of these experiments are discussed in the following.

2.1 Experimental Setup

The diversity of MIP and MINLP and the performance variability of state-of-the-art solvers asks for a careful methodology when measuring performance differences between solver versions. The experimental setup used during the SCIP development process is described in detail in the release report for SCIP 5.0 [35]. This process continues to be used; however, there have been some updates since the last release that will be documented here.

Since the SCIP 6.0 release, MIPLIB 2017 [37] has been released. As a result, the base testset for MIP evaluation was updated. Previously, it consisted of the union of MIPLIB 3, 2003, and 2010 instance collections [51] as well as the COR@L testset [20]. The updated base testset also includes instances from MIPLIB 2017. In order to achieve a good distribution of different MIP problems without overrepresenting particular applications, the selection methodology used to select the MIPLIB 2017 collection from the set of submissions was also utilized to select a base MIP testset for SCIP consisting of 514 instances.

For MINLP, the base testset consists of 200 instances that were manually selected from MINLPLib [64], filtering overrepresented classes and numerically troublesome instances.

Both testsets are restricted to a set of "solvable" instances to reduce the computational effort spent for regular performance testing. The solvable subsets are obtained by selecting only those instances from the main set that could be solved by previous releases or selected intermediate development versions with any of five different random seeds. Performance reported in the following chapter often refers to the solvable testsets, which gives a good estimate for the overall set. In particular, it should rather underestimate the positive effect, if there is any, as instances that can only be solved with the new feature are missing in the solvable subset, while those that are only solved with previous versions are contained. For the comparison between SCIP 6.0 and SCIP 7.0 presented in this section, however, the complete testsets were used.

In order to reduce the impact of performance variability [51], each instance is solved five times with different random seed initializations, including seed zero, with which SCIP is released. For MINLP, where the random seed has less impact, the permutation seed is changed, resulting in a permutation of the problem constraints. In the evaluation, every instance and seed/permutation combination is treated as an individual observation, effectively resulting in testsets containing 2570 MIPs and 500 MINLPs instances. As a result, the term "instance" is often used when actually referring to an

instance-seed-combination during the discussion of computational results, for example, when comparing the number of solved instances. Note that for MINLP, an instance is considered solved when a relative primal-dual gap of 0.0001 is reached; for MIP we use gap limit zero.

Instances for which solver versions return numerically inconsistent results are excluded from the analysis. Besides the number of solved instances, the main measures of interest are the shifted geometric means of the solving times and the branch-and-bound node count. The *shifted geometric mean* of values $t_1, \ldots, t_n$ is

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

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

As can be seen in Tables 1 and 2, these statistics are displayed for several subsets of instances. The subset "affected" filters for instances where solvers show differing 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 subsets "both-solved" and "diff-timeout" contain the instances that can be solved by both of the versions and by exactly one of the versions, respectively. Additionally, MIP results are compared for the subsets of benchmark instances from MIPLIB 2010, MIPLIB 2017, and the COR@L testset, which have a small overlap; MINLP results are reported for the subsets of MINLPs containing "integer" variables and purely "continuous" NLPs.

The experiments were performed on a cluster of computing nodes equipped with Intel Xeon Gold 5122 CPUs with 3.6 GHz and 92 GB main memory. Both versions of SCIP were built with GCC 7.4 and use SoPlex as underlying LP solver: version 4.0.0 (released with SCIP 6.0) and version 5.0.0 (released with SCIP 7.0). SCIP 7.0 uses PaPILO 1.0 to enhance its presolving capabilities. Further external software packages linked to SCIP include the NLP solver Ipopt 3.12.13 [45] built with linear algebra package MUMPS 4.10 [6], the algorithmic differentiation code CppAD [19] (version 20180000.0), and the graph automorphism package bliss 0.73 [46] for detecting MIP symmetry (with a patch applied that can be downloaded from `scip.zib.de`). The time limit was set to 7200 seconds for MIP and to 3600 seconds for the MINLP runs.

2.2 MIP Performance

Table 1 presents a comparison of SCIP 7.0 and SCIP 6.0 with respect to MIP performance. Overall, SCIP 7.0 is about 14% faster than SCIP 6.0. This number, however, is dampened by the fact that more than 40% of the instances in the overall MIP testset cannot be solved by any of the two SCIP versions within the time limit. When considering only instances that can be solved by at least one of the versions (see the [0,7200] bracket), which gives a clearer picture of the actual speedup, a speedup of 24% can be observed. The "affected" instance set is by definition a subset of the [0,7200] bracket. It is worth noting that 97% of the instances are affected and consequently, the speedup on the "affected" subset is the same as for the [0,7200] bracket. On the subset of harder instances in the [100,7200] bracket, SCIP 7.0 is even 31% faster than SCIP 6.0; on the relatively small [1000,7200] bracket, the speedup amounts to 58%.

SCIP 7.0 solves 27 more instances compared to SCIP 6.0. Additionally, the "diff-timeout" subset shows a larger speedup of more than 80%, demonstrating that the 75 instances solved only by SCIP 7.0 are on average solved much faster than the 48 instances solved only by SCIP 6.0. But also on the "both-solved" subset a speedup of 20% can be observed, while the average tree size on this subset stays almost the same.

Table 1: Performance comparison of SCIP 7.0 versus SCIP 6.0 on the complete MIP testset using five different seeds.

	-	SCIP 7.0.0+SoPlex 5.0.0			SCIP 6.0.0+SoPlex 4.0.0			relative	
Subset	instances	solved	time	nodes	solved	time	nodes	time	nodes
all	2557	1468	793.1	5217	1441	901.1	4679	1.14	0.90
affected	1476	1428	189.3	3160	1401	234.5	3147	1.24	1.00
[0,7200]	1516	1468	173.7	2874	1441	215.7	2863	1.24	1.00
[1,7200]	1468	1420	202.9	3213	1393	253.7	3204	1.25	1.00
[10,7200]	1372	1324	264.5	3972	1297	338.6	3984	1.28	1.00
[100,7200]	1023	975	592.6	8029	948	778.9	8038	1.31	1.00
[1000,7200]	508	460	1535.8	21040	433	2431.8	23695	1.58	1.13
diff-timeout	123	75	2902.4	38468	48	5254.0	51519	1.81	1.34
both-solved	1393	1393	135.3	2272	1393	162.5	2202	1.20	0.97
MIPLIB 2010	435	397	283.7	6175	378	345.5	5838	1.22	0.95
MIPLIB 2017	1195	606	1107.5	6811	617	1253.1	5712	1.13	0.84
COR@L	579	393	334.7	2264	375	408.7	2232	1.22	0.99

When considering individual testsets, a 22% speedup is achieved on instances from the COR@L set and the MIPLIB 2010 benchmark set, while the MIPLIB 2017 benchmark set is solved 13% faster.

2.3 MINLP Performance

The focus of the SCIP 7.0 release is on MIP improvements; most work on MINLP was spent for a rewrite of the MINLP constraint handler concept that will only be included in the next SCIP release.

Nevertheless, SCIP 7.0 still provides a speedup on the MINLP testset, see Table 2. All failing instances were excluded, for example instances for which one of the versions returned a solution that was not feasible in tolerances or that a version could not solve (note that the testset contains instances that SCIP 6.0 and SCIP 7.0 cannot solve, but that the MINLP development branch is able to solve). A speedup of 8% can be observed on the overall test set. When excluding instances that none of the versions could solve (bracket [0,3600]), the speedup increases to 18%; for hard instances (bracket [100,3600]), a speedup of 36% is achieved.

SCIP 7.0 solves 18 instances that SCIP 6.0 cannot solve in the time limit, while the reverse holds for 7 instances. On the subset of instances that only one version can solve, SCIP 7.0 is faster than SCIP 6.0 by a factor of more than 6, demonstrating that SCIP 7.0 solves many of the additional instances well before the time limit.

Table 2: Performance comparison of SCIP 7.0 versus SCIP 6.0 on the MINLP testset using five different permutations.

		SCIP 7.0.0+SoPlex 5.0.0			SCIP 6.0.0+SoPlex 4.0.0			relative	
Subset	instances	solved	time	nodes	solved	time	nodes	time	nodes
all	919	460	218.3	3674	456	236.3	4932	1.08	1.34
affected	427	420	16.9	1319	416	20.2	2007	1.19	1.52
[0,3600]	467	460	13.6	1031	456	16.1	1524	1.18	1.48
[1,3600]	338	331	33.6	2422	327	41.8	4063	1.24	1.68
[10,3600]	232	225	89.4	4473	221	115.3	8532	1.29	1.91
[100,3600]	129	122	263.4	8909	118	357.8	24091	1.36	2.70
[1000,3600]	56	49	732.9	27254	45	1182.7	77255	1.61	2.83
diff-timeout	18	11	244.3	4746	7	1571.5	6559	6.43	1.38
both-solved	449	449	12.0	967	449	13.2	1435	1.10	1.48
continuous	239	74	420.1	1777	74	428.5	1483	1.02	0.83
integer	665	371	191.1	5065	367	210.6	8122	1.10	1.60

3 PaPILO

The constraint-based view on the problem of SCIP allows for great flexibility and is a major feature that enables SCIP to seamlessly integrate the handling of nonlinear constraints. When it comes to MIP problems, however, it deprives SCIP of a global view on the constraint matrix of the problem. Many presolve reductions for MIP problems involve not only information of a single constraint, but of several rows or columns of the constraint matrix. Acquiring this kind of information within SCIP is sometimes not possible or too expensive for an expedient implementation of certain methods.

For this reason, the development of PaPILO, short for "Parallel Presolve for Integer and Linear Optimization", has been initiated. PaPILO is a C++ software package for presolving MIP problems, which is a less general class of problems than the problems that can be solved by SCIP. This allows PaPILO to use tailored data structures for holding a problem, particularly a row-major and column-major copy of the constraint matrix. Moreover, PaPILO was designed to facilitate the exploitation of modern parallel hardware and achieves this in a way that is unique for this type of software. Another distinguishing feature is the multi-precision support via the use of C++ templates. This allows the presolving routines to use exact rational arithmetic, or extended-precision floating point arithmetic.

The release of PaPILO 1.0 does not implement any novel presolve reductions. The implemented reductions are: coefficient tightening, constraint propagation, singleton stuffing, dual fixing, removal of parallel rows and columns, substitution of (implied) free variables, implied integer detection, dual reductions using the complementary slackness conditions (see Section 4.1.3), probing, fixing of dominated columns, constraint sparsification. For more information on those presolve reductions the reader is referred to Achterberg et al. [5].

3.1 A Deterministic Parallelization Scheme for Data and Task Parallelism

PaPILO exploits both data and task parallelism during presolving. A unique feature is that multiple presolving rules can always be executed in parallel, because they receive read-only access to all problem data. The presolvers scan this data for possible reductions and return them to the core of the library.

Due to this approach, a reduction might be applied to a different state of the problem than it was derived from. It is difficult to guarantee correctness under these circumstances, because a reduction found by one presolver might interfere with a reduction of another presolver. To remedy this situation, the presolvers return the reductions embedded within a transaction as it is done in database systems [10]. A transaction contains a sequence of steps that can lock or reduce parts of the problem. The steps within the transaction are composed in a way that guarantees correctness if the locked parts have not been modified.

As an illustrative example, consider the following rows and bounds as part of some problem.

$$\begin{aligned} &\dots \\ x + y &\geq 1 \qquad \text{(row 1)} \\ x + y + z &= 1 \qquad \text{(row 2)} \\ &\dots \\ x, y, z &\in [0, 1] \end{aligned}$$

Here, x is a so-called implied free variable because the lower bound of x is implied by row 1 and the upper bound of x is implied by row 2. Hence, the variable bounds on x

can be dropped and the variable x can be substituted via $x = 1 - y - z$ and row 2 can be removed. The transaction to communicate the substitution of x within PaPILO would therefore consist of the following steps:

1. Lock the bounds of x
2. Lock row 1 (implies $x \geq 0$)
3. Lock row 2 (implies $x \leq 1$ and is used for substitution)
4. Substitute x using row 2

Together with the assumption that in PaPILO variable bounds are never relaxed, the first three steps of the transaction are sufficient to guarantee that x is still an implied free variable and the equation row 2 can be used to substitute it, regardless of other transactions that are applied before or after. If one of the locks in steps 1 to 3 are violated, the transaction is discarded.

Database systems use this approach to run multiple queries in parallel while guaranteeing a consistent state of the data. In PaPILO the approach allows presolvers to propose reductions for a fixed copy of the problem that can still be applied after some modifications have been made. This strategy allows presolvers to use the same copy of the problem to search for reductions in parallel. Subsequently, PaPILO processes the transactions sequentially, in a deterministic order, and might discard some of them.

Since the search for reductions is usually the expensive part of presolving routines, compared to the modification of the problem that is typically cheap, the transaction approach is expected to bring improved efficiency from parallelization. When data in rows or columns is modified while a transaction is applied they are marked as such. Before a transaction is applied all its locks are checked against the recorded states of the rows and columns and the transaction is discarded if a lock is violated. The result after applying the transactions only depends on the order in which the transactions are applied, and not on the order they are found. As the reductions are applied in a fixed order, the routine gives deterministic results regardless of the number of threads that were used.

In addition to running multiple presolvers in parallel, some presolvers themselves can exploit parallelization. This is currently the case for probing, detecting parallel rows and columns, identifying dominated columns, and performing constraint sparsification. All the parallel presolvers exploit data parallelism and sort all non-deterministically ordered results before returning them.

This recursive nesting of data and task parallelism is effective when a suitable runtime manager controls the load distribution. For this purpose the Intel TBB library [44] is linked, which uses heuristics that check the load of the system at runtime when deciding whether data parallel tasks should be further split or be processed sequentially by the current thread.

The effectiveness of this parallelization scheme can be observed especially well on instances that spent significant time in presolving. One example of such an instance is `ex10` of MIPLIB 2017, which is solved during presolving mostly by probing. The speedup in solving time for different numbers of threads using SCIP 7.0 with PaPILO 1.0 is depicted in Figure 1. One can see that the speedup is almost linear when assuming that about 5% of the time is spent in sequential code. For up to 8 threads the speedup is almost perfectly linear. Only for a bigger number of threads some amount of overhead associated with a per-thread setup cost becomes visible.

3.2 Integration with SCIP

SCIP 7.0 includes a new presolver plugin that calls PaPILO if the problem being solved is a MIP. To use an external presolve routine within SCIP, the plugin needs to convert

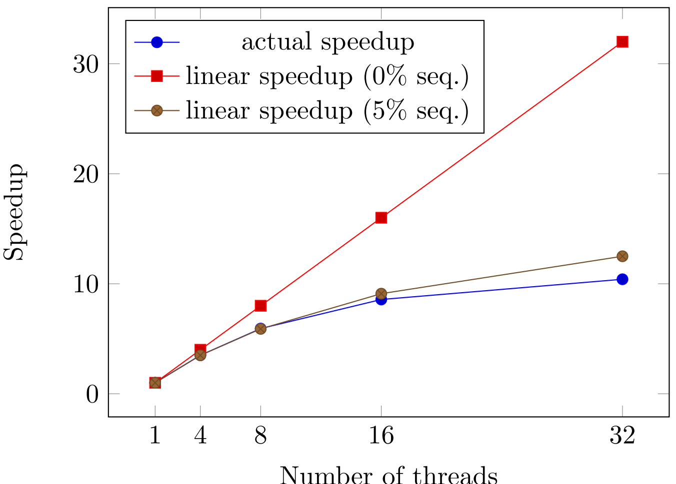


Figure 1: Speedup of `ex10` using SCIP 7.0 with PaPILO 1.0 for different numbers of threads. Additionally shows ideal linear speedup curves when 0% and 5% of the time are spent inside sequential code.

SCIP's problem representation into matrix form to call PaPILO and susequently apply the reductions found by PaPILO in a way that allows SCIP to convert solutions back into the original problem space. Reductions of the variable space, including fixings and substitutions, can be communicated to SCIP based on the postsolve information returned by PaPILO. Changes to the constraint-matrix, on the other hand, require a tighter integration to be communicated seamlessly into SCIP's problem representation. In the current version of the presolver this is implemented by deletion of all constraints and subsequent recreation from PaPILO's reduced problem. Therefore, SCIP 7.0, by default, only communicates contraint modifications when the number of constraints or nonzeros decreased by at least 20% due to PaPILO's presolve. The removal of parallel columns is currently not executed when PaPILO is called from SCIP. The reason is that the necessary treatment in postsolve is not yet implemented in SCIP 7.0.

The performance impact within SCIP 7.0 is notable even when PaPILO runs in sequential mode. Disabling the presolver plugin for PaPILO yields a 6% slowdown on the internal MIP benchmark testset.

4 Advances in SCIP

The new 7.0 release of SCIP provides significant improvements for mixed-integer programming, both for the classical branch-and-cut approach that SCIP applies by default and for decomposition approaches like Benders' decomposition. The most important changes are described in the following.

4.1 Presolve

The improvements to presolving in SCIP 7.0 has lead to the development of three methods: LP-based bound tightening on two constraints, two-column nonzero cancellation and exploiting complementary slackness. These three methods are based on using pairs of constraints or variables to identify reductions. Further, a new presolver has been added to enable calling PaPILO from within SCIP, see Section 3 for more details.

4.1.1 LP-based Bound Tightening on Two Constraints

The idea of using two rows at once for bound tightening or recognizing redundant constraints has already been presented by Achterberg et al. [5]. Let two constraints be given as follows:

$$\begin{array}{ccccccc} A_{rU}x_U & + & A_{rV}x_V & & & \geq & b_r, \\ A_{sU}x_U & + & & & A_{sW}x_W & \geq & b_s. \end{array}$$

Then one can consider the following single-row LPs where the variables appearing in both constraints form the objective:

$$\begin{aligned} y_{\max} &= \max\{A_{rU}x_U \;:\; A_{sU}x_U + A_{sW}x_W \geq b_s\}, \\ y_{\min} &= \min\{A_{rU}x_U \;:\; A_{sU}x_U + A_{sW}x_W \geq b_s\}. \end{aligned} \tag{3}$$

It has been shown by Dantzig [21] and Balas [9] that these single-row LPs can be solved in linear time. Let $j \in V$, so x_j is not contained in the overlap and write $V' = V \setminus \{j\}$. Depending on the sign of a_{rj} one can now derive possibly stronger bounds via

$$\begin{aligned} x_j &\geq \tfrac{b_r - \sup(A_{rV'}x_{V'}) - y_{\max}}{a_{rj}} \quad \text{for } a_{rj} > 0, \\ x_j &\leq \tfrac{b_r - \sup(A_{rV'}x_{V'}) - y_{\max}}{a_{rj}} \quad \text{for } a_{rj} < 0. \end{aligned} \tag{4}$$

Similarly, a constraint is redundant if the following condition holds:

$$\inf(A_{rV}x_V) \geq b_r - y_{\min}. \tag{5}$$

Hashing Mechanism for Finding Row Pairs Since short runtimes are mandatory for presolve methods, it is usually too expensive to apply reductions to each row pair in a given problem. For the presolve method presented above, it is important to find row pairs such that a large number of variables have coefficients of opposing sign in the two constraints. One approach is to scan through the problem and create four hashlists L_{++}, L_{--}, L_{+-}, L_{-+}, one for each combination of signs of two variable coefficients. Given a hash function H to hash pairs of variable indices, these lists then contain tuples (h, r) consisting of a hash value h and a row index r. For the two pairs L_{++}, L_{--} and L_{+-}, L_{-+}, the two-row bound tightening methods are applied to all pairs of rows r, s with $(h, r) \in L_{++}$, $(h, s) \in L_{--}$ or $(h, r) \in L_{+-}$, $(h, s) \in L_{-+}$ as for these row-pairs, the hash-values h coincide and, unless a hash collision occured, there exist at least two variables with opposing coefficient signs. To further limit runtime, several working limits are used in our implementation. More precisely, the size of the four hashlists, the total number of variable pairs considered are limited and the method stops after a limited number of consecutive row-pairs evaluations without bound improvement. To reduce redundant computations, our implementation uses a hashtable to check if a row pair has already been processed before doing so. The method also stops after finding too many consecutive already processed row pairs.

Chen et al. [18] evaluated the performance on the MIPLIB 2017 Benchmark Set. Using the LP-based bound tightening resulted in an overall neutral performance impact with positive impact of up to 4 % on instances in the subset [1000,7200].

4.1.2 Two-Column Nonzero Cancellation

In SCIP 5.0 [35], a two-row nonzero cancellation presolver had been added. This presolver now transfers the idea to columns. More precisely, consider the sum of two columns

$$\begin{bmatrix} A_{Uj} \\ A_{Vj} \\ A_{Wj} \\ \end{bmatrix} x_j + \begin{bmatrix} A_{Uk} \\ A_{Vk} \\ \\ A_{Yk} \end{bmatrix} x_k, \tag{6}$$

where $j,\, k \in \mathcal{N}$ and $U,\, V,\, W,\, Y \subseteq \mathcal{M}$ are disjoint subsets of the row indices. Suppose there exists a scalar $\lambda \in \mathbb{R}$ such that $A_{Uk} - \lambda A_{Uj} = 0$ and $A_{Vk} - \lambda A_{Vj} \neq 0$. In the case of a continuous variable x_j one can rewrite (6) as

$$\begin{bmatrix} A_{Uj} \\ A_{Vj} \\ A_{Wj} \\ \\ \end{bmatrix} (x_j + \lambda x_k) \ + \ \begin{bmatrix} \\ A_{Vk} - \lambda A_{Vj} \\ -\lambda A_{Wj} \\ A_{Yk} \end{bmatrix} x_k,$$

and introducing a new variable $z := x_j + \lambda x_k$ yields

$$\begin{bmatrix} A_{Uj} \\ A_{Vj} \\ A_{Wj} \\ \\ \end{bmatrix} z \ + \ \begin{bmatrix} \\ A_{Vk} - \lambda A_{Vj} \\ -\lambda A_{Wj} \\ A_{Yk} \end{bmatrix} x_k. \tag{7}$$

It follows that the lower and upper bounds on z are given by

$$\ell_z = \begin{cases} \ell_j + \lambda \ell_k, & \text{for } \lambda > 0, \\ \ell_j + \lambda u_k, & \text{for } \lambda < 0, \end{cases} \quad \text{and} \quad u_z = \begin{cases} u_j + \lambda u_k, & \text{for } \lambda > 0, \\ u_j + \lambda \ell_k, & \text{for } \lambda < 0, \end{cases}$$

respectively. However, the constraint

$$\ell_j \leq z - \lambda x_k \leq u_j \tag{8}$$

needs to explicitly be added to keep the bounds $\ell_j \leq x_j \leq u_j$. Due to this additional constraint, the difference in the number of nonzeros $|U| - |W|$ must be larger than in the two-row version. Full details and the exact procedure for integral variables x_j as well as a description of the hashing mechanism to find suitable column pairs is provided by Chen et al. [18].

Chen et al. [18] evaluated the performance on the MIPLIB 2017 Benchmark Set. In total, its impact can be considered neutral with two more instances being solved at the cost of a slightly prolonged runtime for instances in the subset [1000,7200].

4.1.3 Exploiting Complementary Slackness on Two-Columns of Continuous Variables

A first version of this presolve method was previously implemented in SCIP 4.0 [60]. The basic idea is to consider only those columns of the mixed-integer optimization problem that belong to continuous variables and carry out a bound tightening on these columns to determine bounds for the dual variables. The bounds of the dual variables can then be used to apply implications of the complementary slackness theorem (see Schrijver [72]), i.e., to fix variables or to determine that inequality constraints are actually equations in an optimal solution.

If only individual columns are considered, this is usually very fast, but the bounds for the dual variables are often weak. A reasonable compromise to be still fast but find tighter bounds is to look at two continuous columns simultaneously. To achieve this, one can simply reuse the hashing mechanism for finding row pairs from Section 4.1.1 and apply them to two columns. More details on the implementation of this presolve method are given by Chen et al. [18].

Chen et al. [18] evaluated the performance of this presolve method on the MIPLIB 2017 Benchmark Set. The presolve method performs reductions on only 11 instances and the overall impact is neutral. This is not necessarily due to the inefficiency of the reductions themselves, but rather a consequence of the very low number of instances where this method can be applied, because of missing continuous variables. However, on the real-world supply chain instances used by Schewe et al. [71] a larger influence of this presolve method is observed.

4.2 Decomposition Structure

Most MIPs have sparse constraint matrices in the sense that the vast majority of columns and rows have only very few nonzero entries. Due to sparsity, a (bordered) block-diagonal form might be obtained by permuting the rows/columns of the matrix. Identifying such a form allows for potentially rendering large-scale complex problems considerably more tractable. Solution algorithms could be designed exploiting the underlying structure and yielding smaller problems that are significantly easier to handle. In this sense, a decomposition identifies subproblems (subsets of rows and columns) that are only linked to each other via a set of linking rows and/or linking columns (border components), but are otherwise independent. The special case of completely independent subproblems (with no linking rows and columns), for example, can be handled by solving the much smaller subproblems and concatenating their optimal solutions. This case has already been integrated into SCIP as a successful presolving technique [31] within the default plugin `cons_components.c`.

For $k \geq 0$, a partition $\mathcal{D} = (D^{\text{row}}, D^{\text{col}})$ of the rows and columns of the constraint matrix A into $k+1$ pieces each,

$$D^{\text{row}} = (D_1^{\text{row}}, \dots, D_k^{\text{row}}, L^{\text{row}}), \quad D^{\text{col}} = (D_1^{\text{col}}, \dots, D_k^{\text{col}}, L^{\text{col}}),$$

is called a *decomposition* of A if $D_q^{\text{row}} \neq \emptyset$, $D_q^{\text{col}} \neq \emptyset$ for $q \in \{1, \dots, k\}$ and it holds for all $i \in D_{q_1}^{\text{row}}$, $j \in D_{q_2}^{\text{col}}$ that $a_{i,j} \neq 0$ implies $q_1 = q_2$. The special rows L^{row} and columns L^{col}, which may be empty, are called linking rows and linking columns, respectively. In other words, the inequality system $Ax \geq b$ can be rewritten with respect to a decomposition $\mathcal{D}$ by a suitable permutation of the rows and columns of A as the following equivalent system

$$\begin{pmatrix} A_{[D_1^{\text{row}}, D_1^{\text{col}}]} & 0 & \cdots & 0 & A_{[D_1^{\text{row}}, L^{\text{col}}]} \\ 0 & A_{[D_2^{\text{row}}, D_2^{\text{col}}]} & 0 & 0 & A_{[D_2^{\text{row}}, L^{\text{col}}]} \\ \vdots & 0 & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & A_{[D_k^{\text{row}}, D_k^{\text{col}}]} & A_{[D_k^{\text{row}}, L^{\text{col}}]} \\ A_{[L^{\text{row}}, D_1^{\text{col}}]} & A_{[L^{\text{row}}, D_2^{\text{col}}]} & \cdots & A_{[L^{\text{row}}, D_k^{\text{col}}]} & A_{[L^{\text{row}}, L^{\text{col}}]} \end{pmatrix} \begin{pmatrix} x_{[D_1^{\text{col}}]} \\ x_{[D_2^{\text{col}}]} \\ \vdots \\ x_{[D_k^{\text{col}}]} \\ x_{[L^{\text{col}}]} \end{pmatrix} \geq \begin{pmatrix} b_{[D_1^{\text{row}}]} \\ b_{[D_2^{\text{row}}]} \\ \vdots \\ b_{[D_k^{\text{row}}]} \\ b_{[L^{\text{row}}]} \end{pmatrix}, \tag{9}$$

where we use the shorthand syntax $A_{[I,J]}$ to denote the $|I|$-by-$|J|$ submatrix that arises from the deletion of all entries from A except for rows I and columns J, for nonempty row and column subsets $I \subseteq \{1, \dots, m\}$ and $J \subseteq \{1, \dots, n\}$.

Structure Creation With SCIP 7.0, it is now easier to pass user decompositions to SCIP that can be used within the BD framework or by user algorithms (see Section 4.3 and 4.9 for more details about the applications). A decomposition structure can be created using the SCIP-API, specifying whether the decomposition belongs to the original or transformed problem and the number of blocks. The variables and/or constraints can be assigned labels by the user either one-by-one, or in batches. It is possible to complete the decomposition, or ensure that it is internally consistent, by calling the automatic label computation procedures. Alternatively, SCIP also provides a file reader for decompositions in constraints of the original problem. The standard file extension for this file format is ".dec", which is also supported by GCG. In what follows, the notion *block* is used to define the submatrix $A_{[D_q^{\text{row}}, D_q^{\text{col}} \cup \bar{L}^{\text{col}}]}$ for $q \in \{1, \dots, k\}$, $\bar{L}^{\text{col}} \subseteq L^{\text{col}}$, where it holds for all $i \in D_q^{\text{row}}, j \in L^{\text{col}}$ that $a_{i,j} \neq 0$ implies $j \in \bar{L}^{\text{col}}$.

The user decomposition file specifies the number of blocks in the decomposition, their corresponding unique labels, the constraints (by their names) belonging to each block as well as those identified as linking constraints.

Upon reading a valid decomposition file, a decomposition structure is created, in which the corresponding variable labels are inferred from the constraint labels, giving precedence to block over linking constraints. Therefore, a variable is assigned

– the label of its unique block, if it only occurs in exactly one named block, and possibly in the linking constrains;
– the special label of a linking variable if it occurs only in the linking constraints or in two or even more named blocks.

The created structure is eventually added to the decomposition storage of SCIP for the current problem, as each problem could have several applicable decompositions with varying characteristics. Additionally, it is possible to override the existing labels of the constraints, while inferring the new labels from the variable labels.

An alternative variable labeling is required if the supplied decompositions will be used for the application of BD. SCIP is informed to use this alternative labeling by setting the parameter `decomposition/benderslabels` to `TRUE`, or by passing the corresponding flag when calling the decomposition constructor `SCIPcreateDecomp()`. The labeling then proceeds as follows: Initially, all variables are unlabeled, and then receive a label as "master only" or "block" if they are encountered in a linking constraint or block constraint, respectively. The deviation from the default labelling occurs if the variable is encountered in two or more constraints. If a master only variable is encountered in a block constraint, or a block variable is encountered in a linking constraint or a constraint from a different block, then these variables are re-labeled as linking.

Due to variables' fixing or redundancy detection during presolving, the constraints could be modified—or possibly deleted—in the transformed problem leading certain decomposition blocks to become empty. Additionally, in the context of bound strengthening, some constraints could get merged, resulting in potential decomposition blocks' merging. Therefore, the constraints' labeling must be triggered again after presolving. When forming the transformed problem, SCIP automatically transforms all user decompositions at the beginning of the root node based on the variables' labels.

Decomposition Statistics The SCIP-API also provides the possibility to compute relevant decomposition statistics in order to offer useful insights to the quality of the considered problem's decomposition or deduce correlations between certain aspects in the structures and the performance of the invoked user algorithms. In particular, when the labeling process is concluded, the following quantities are computed and displayed:

– number of blocks;
– number of linking variables and linking constraints;
– size of the largest and smallest block regarding the number of constraints;
– *Area score:* The area score (see also [36], Sec. 5) is used by GCG to rank decompositions during the automatic detection procedure. For a decomposition $\mathcal{D} = (D^{\text{row}}, D^{\text{col}})$, the area score is defined as

$$\text{areascore}(D) = 1 - \frac{\sum_{b=1}^{k} |D_b^{\text{row}}||D_b^{\text{col}}| + n|L^{\text{row}}| + m|L^{\text{col}}| - |L^{\text{row}}||L^{\text{col}}|}{mn} \;.$$

  In the case of a MIP, the area score intuitively measures the coverage of the rearranged matrix (9) by 0's. Decompositions with few linking variables and/or constraints and many small blocks $A_{[D_b^{\text{row}}, D_b^{\text{col}}]}$ will have an area score close to 1, whereas coarse decompositions of a matrix have smaller area scores. The trivial decomposition with a single block has the worst possible area score of 0.

– *Modularity:* This measure is used to assess the quality of the *community structure* within a decomposition. In the context of graph theory, *communities* can be defined

as sets of vertices having inner dense connections. Khaniyev et al. [49] describe the modularity quantity as the fraction of the edges in the graph that connect vertices of the same type (within-community edges) minus the expected value of the same quantity in a graph with random connections between vertices. Similarly, we use the modularity quantity to assess the inner connectedness and density at the level of the blocks in a certain decomposition and, hence, their independence and the potential of their separation from the rest of the blocks. More precisely, we identify the presence of an inner block connection (an inner edge) through the presence of a variable in a constraint, both—the variable and the constraint—belonging to the same block. The modularity of the decomposition is computed as follows:

$$\sum_{i=1}^{k} \frac{e_i}{m}\left(1 - \frac{e_i}{m}\right),$$

where k is the number blocks, e_i is the number of inner edges within block i and m is the total number of edges.

– *Block graph statistics:* A block graph is constructed with the aim of depicting the connection between the different blocks in a decomposition through the existing linking variables in the constraints. Note that the linking constraints are intentionally skipped in this computation. $G = (V, E)$ denotes a block graph with vertex set V and edge set E. Each vertex in the graph represents a block in the decomposition; $V = \{v_1, \dots, v_k\}$. An edge $e = \{v_s, v_t\}$ is added to G, if and only if there exists a column $\ell \in L^{col}$, a row $i \in D_s^{row}$ and a row $j \in D_t^{row}$, such that $a_{i,\ell} \neq 0$ and $a_{j,\ell} \neq 0$. From the constructed graph, we compute the number of edges, articulation points and connected components, as well as the maximum and minimum degree. Note that building the block graph can become computationally expensive with large and dense decompositions. Thus, it is possible through the user parameter `decomposition/maxgraphedge` to define a maximum edge limit. The construction process will be interrupted once this limit is reached, in which case only approximate estimations of the block graph statistics will be displayed and accompanied with a warning message.

4.3 Primal Heuristics

SCIP 7.0 introduces two new heuristics and an update to the GINS heuristic. A common topic for two of the three heuristics is the exploitation of decomposition information.

4.3.1 Adaptive Diving

Diving heuristics explore an auxiliary path of the tree in a depth-first fashion. A description of most of the available diving heuristics in SCIP can be found in [2]. With the release of SCIP 7.0 an adaptive diving heuristic extends the diving heuristics of SCIP. Rather than defining an individual diving strategy, adaptive diving collects all diving strategies defined by other primal heuristics, including user-defined ones. In the same spirit as adaptive Large Neighborhood Search [38], adaptive diving learns during the search which of the different diving strategies are most successful. Each time adaptive diving is executed, it prefers strategies that are either uninitialized or that have shown to yield useful conflict information. A description of adaptive diving and computational results can be found in [39].

On the technical side, adaptive diving is implemented as a primal heuristic plugin `heur_adaptivediving.c` in SCIP. Statistics that were collected within adaptive diving are stored separately from the statistics of an individual diving heuristic, but can be

queried together. Finally, user diving strategies can now be declared as *private* to exclude them from adaptive diving.

4.3.2 Penalty Alternating Direction Method

With SCIP 7.0 the new construction heuristic `padm` was added. This heuristic explicitly requires a decomposition provided by the user via the new decomposition structure. A detailed description of penalty alternating direction methods can be found in Geißler et al. [34]. These methods were used as a primal heuristic for supply chain problems in Schewe et al. [71].

The heuristic splits a general MINLP into several subproblems according to the user decomposition, whereby the linking variables get copied and their difference between copies is penalized. Linking constraints cannot be handled, but an attempt is made to assign them to one of the blocks.

The first subproblem of iteration p can be written as

$$\begin{aligned}
\min \quad & \sum_{i\in L^{\mathrm{col}}} \sum_{b\in\{2,\dots,k\}} \mu_i^{b,+} s_i^{b,+} + \mu_i^{b,-} s_i^{b,-} \\
\text{s.t.} \quad & g_k(x, z^1) \le b_k && \text{for all } k \in \mathcal{M}^1, \\
& z_i^1 + s_i^{b,+} - s_i^{b,-} = z_i^{b,p} && \text{for all } i \in L^{\mathrm{col}}, b \in \{2,\dots,k\}, \\
& \ell_i \le x_i \le u_i && \text{for all } i \in \mathcal{N}^1, \\
& \ell_i \le z_i^1 \le u_i && \text{for all } i \in L^{\mathrm{col}}, \\
& x_i, z_i^1 \in \mathbb{Z} && \text{for all } i \in \mathcal{I} \cap (\mathcal{N}^1 \cup L^{\mathrm{col}}), \\
& s_i^{b,+}, s_i^{b,+} \in \mathbb{R}_+ && \text{for all } i \in L^{\mathrm{col}}, b \in \{2,\dots,k\}.
\end{aligned} \tag{10}$$

The sets $\mathcal{M}^1$ and $\mathcal{N}^1$ contain all constraints or variables of the first block, index b specifies the other blocks. The slack variables $s_i^{b,+}$ and $s_i^{b,-}$ represent the difference between two copies of the linking variables z_i.

The main part of the algorithm is organized in two loops: In the inner loop, the subproblems are solved on an alternating basis until they arrive at the same values of the linking variables. If they do not reconcile after a couple of iterations (controlled by the user parameter `heuristics/padm/admiterations`), the penalty parameters μ are increased in the outer loop and the subproblems are solved again on an alternating basis.

The subproblems can be warmstarted by using the old solution. Let $(x, z, s)^{p-1}$ be the solution of the last iteration of block q and $z_i^{b,p}$ be the current assignment of the linking variables. Then $(x^{p-1}, z^{p-1}, \tilde{s})$ with

$$\begin{aligned}
& \tilde{s}_i^{b,+} := \max\{z_i^{b,p} - z_i^{p-1},\ 0\} \\
\text{and} \quad & \tilde{s}_i^{b,-} := \max\{z_i^{p-1} - z_i^{b,p},\ 0\} \quad \text{for all } i \in L^{\mathrm{col}}, b \in \{1,\dots,k\} \setminus \{q\}
\end{aligned}$$

is a feasible solution of block q in iteration p and can be used as start solution.

As per default the heuristic `padm` is called before the root node and the linking variables $z_i^{b,0}$ are initialized with zero. By changing the parameter `heuristics/padm/timing` it can also be called after the root node where the linking variables are initialized with the LP solution value.

If a suitable decomposition is available, preliminary results on MIPs showed that the heuristic finds a feasible solution in 62% of the cases.

4.3.3 GINS Adjustments to Exploit Decomposition Information

Since version 4.0 [60], SCIP has featured graph induced neighborhood search. This Large Neighborhood Search heuristic explores an auxiliary problem corresponding to

a connected subgraph of the bipartite graph $G = (V, E)$ whose nodes correspond to the variables and constraints and whose edges correspond, in the case of a MIP, to the nonzero entries of the constraint matrix A.

The variable neighborhood is selected around a variable node $v \in V$ and contains all variables whose nodes have a breadth-first distance of at most 6 from v in G. Since its introduction, GINS seeks to find such a neighborhood of maximum *potential*, by which we denote the objective difference between the current LP and incumbent solutions, restricted to the neighborhood. A neighborhood with large potential is created by trying out several variable nodes as center, evaluating the potential of its neighborhood, and keeping the neighborhood that maximizes the potential among the neighborhoods tried.

With SCIP 7.0, GINS has been extended to also consider user-provided decomposition labels. Given a decomposition $\mathcal{D} = (D^{\text{row}}, D^{\text{col}})$ of an instance, the heuristic merges variable blocks from the partition $D^{\text{col}} = (D_1^{\text{col}}, \ldots, D_k^{\text{col}}, L^{\text{col}})$ into neighborhoods and processes them in a rolling horizon [53] fashion. Without loss of generality, we assume $D_1^{\text{col}}, \ldots, D_k^{\text{col}}$ are sorted by decreasing potential. From the front, the first block $D_{i^*}^{\text{col}}$ is searched that meets the fixing rate of GINS (66 % by default), i.e., the first block with less than 34 % integer variables. The tentative variable set $U \coloneqq D_{i^*}^{\text{col}}$ is then extended by

1. the linking variables L^{col},
2. further blocks $D_{i^*+1}^{\text{col}}, D_{i^*+2}^{\text{col}}, \ldots, D_{i^*+q}^{\text{col}}$,

as long as U still meets the fixing rate of GINS. GINS then searches an auxiliary problem by fixing all integer variables outside of U to their values in the current incumbent solution. The continuous overlap parameter `heuristics/gins/overlap` controls how many blocks are kept in the tentative variable set between subsequent calls. An overlap of 0.0 means that there is no overlap, i.e.,$D_{i^*+q+1}^{\text{col}}$ is the first considered block, whereas with an overlap of 1.0, only $D_{i^*}^{\text{col}}$ is skipped.

The parameter `heuristics/gins/consecutiveblocks` can be used to set the sorting order for the blocks. The default value of `TRUE` sorts the blocks by increasing label order. Setting this parameter to `FALSE` will sort the blocks by decreasing potential. By default, GINS falls back to its original neighborhood initialization if the decomposition does not yield a suitable neighborhood.

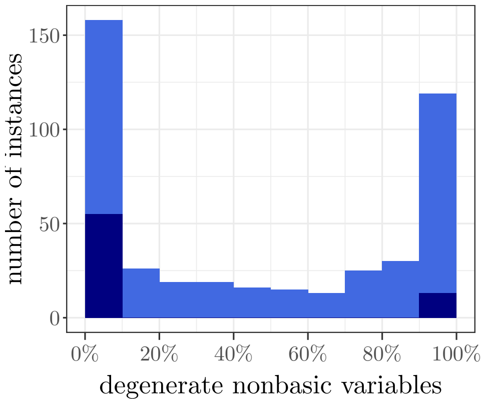
4.4 Improvements in Branching Rules

SCIP 7.0 comes with several improvements in branching, ranging from improvements in its default branching rules for MIP and MINLP to branching rules tailored to specific .

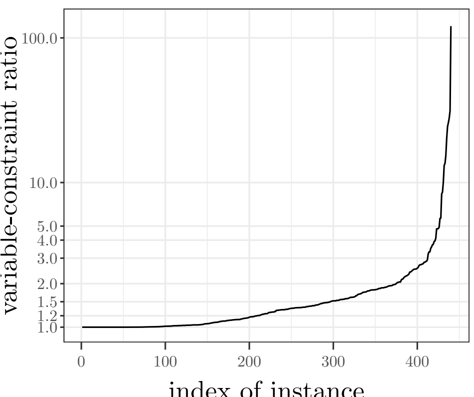
4.4.1 Degeneracy-aware Hybrid Branching

The default hybrid branching rule [4] of SCIP has been extended by the option to adjust the weights of the different scores based on the degree of dual degeneracy in the current LP solution. In a basic LP solution, a nonbasic (structural or slack) variable is called *dual degenerate*, if its reduced cost is zero. As a consequence, such variables can be pivoted into the basis and potentially change their value without changing the LP objective value. Therefore, dual degeneracy typically leads to alternative LP optima being present.

SCIP uses two measures for dual degeneracy: the *degeneracy share of nonbasic variables* and the *variable-constraint ratio of the optimal face*. For an LP relaxation in standard form with m constraints and n variables (possibly including slack variables), let $\bar{n}$ be the number of dual degenerate nonbasic variables. The degeneracy share $\alpha \in [0, 1]$ is defined as $\alpha = \frac{\bar{n}}{n-m}$, where 0 corresponds to no dual degenerate variables and 1 means



(a) Share of nonbasic variables that are dual degenerate. The darker parts represent instances with the extreme degeneracy rates of 0 % and 100 %.



(b) Variable-constraint ratios of the optimal face. The instances are sorted by their non-decreasing ratios.

Figure 2: Dual degeneracy measures for the final root LP (MMMC testset).

all nonbasic variables are dual degenerate. The latter measure may capture the effects of degeneracy better than the former measure in cases where the number of variables is much larger than the number of constraints. In such cases, the degeneracy share may still be small even though the number of nonbasic dual degenerate variables exceeds the basis size drastically. The variable-constraint ratio is 1 if no dual degeneracy is present and can increase up to $\frac{n}{m}$, if all nonbasic variables are dual degenerate. Both measures can be queried via the method `SCIPgetLPDualDegeneracy`.

Figure 2 illustrates these measures of dual degeneracy for the final root LPs of the instances in the MMMC testset consisting of the benchmark sets of MIPLIB 3, 2003, and 2010, and the COR@L testset. Only 55 instances show no dual degeneracy in the final root LP; for 149 instances, 80 % or more of the nonbasic variables are degenerate. 166 instances have a variable-constraint ratio no larger than 1.1; on the other hand, 66 instances have ratios larger than 2.0, one even has a ratio of 120. For a more detailed analysis of dual degeneracy at the root node and throughout the tree, we refer to Gamrath et al. [33].

Computational experiments showed that a high dual degeneracy implies that for each branching candidate, there is a high probability that an alternative optimal LP solution exists where the candidate's value is integer, see Gamrath et al. [33]. In such a case, the dual bound improvement of at least one of the two potential child nodes is zero for each of the candidates. If the exact dual bound improvements would be known, for example, by strong branching, the product score that SCIP uses to combine the improvements of the two child nodes would be close to zero as well. In that case, the other scores used by hybrid branching (conflict, inference, and cutoff score) become more important for the final branching decision. As a consequence, SCIP aims at selecting a branching variable that provides an improvement for the first child as well, if not in dual bound, then in variables being fixed by domain propagation or in moving closer to an infeasible problem.

Degeneracy-aware hybrid branching uses this observation and adjusts the weights based on degeneracy information. To this end, it computes factors ψ and ω based on the degeneracy share α and the variable-constraint ratio β as follows:

$$\psi(\alpha) = \begin{cases} 10^{10(\alpha-0.7)} & \text{if } \alpha \geq 0.8 \\ 1 & \text{else} \end{cases} \tag{11}$$

and

$$\omega(\beta) = \begin{cases} 10\,\beta & \text{if } \beta \geq 2.0 \\ 1 & \text{else.} \end{cases} \tag{12}$$

Then, the weight for dual bound improvement, which is 1 by default, is divided by the product of the two factors. In contrast, the weights for conflict score (0.01 by default), inference, and cutoff scores (both 0.0001 by default) are multiplied by the product. As a result, as soon as the degeneracy share α is at least 80 % or the variable-constraint ratio β is 2.0 or higher, conflict information is weighted at least as much as dual bound improvement. If those degeneracy measures increase, the impact of the dual bound gets smaller and may even fall behind cutoff and inference score.

Additionally, strong branching is disabled completely at the current node if $\psi(\alpha) \cdot \omega(\beta) \geq 10$, that is, if any of the two conditions mentioned before is fulfilled. The motivation for this is that, as other measures are more important now than the dual bound improvement, the additional effort for strong branching is deemed unnecessary. As a consequence, this allows to apply strong branching at a later node where the LP solution is less degenerate if the pseudocosts of the variables did not become reliable until then. Thus, the benefit of this modification is not necessarily the time saved in strong branching, but also that the time for strong branching is spent where it pays off more because the strong branching information is more useful and also provides a better initialization of the pseudocosts. For more details, we refer to Berthold et al. [13].

Degeneracy-aware hybrid branching is enabled by default in SCIP 7.0. Note that the standard hybrid branching is performed at the root node and the degeneracy-aware version is only used afterward. This choice resulted from the observation that strong branching at the root node has the added benefit that infeasible strong branching LPs result in bound changes which often lead to a restart at the root node. Therefore, standard hybrid branching is performed to ensure that these root restarts are still triggered. Performance experiments performed at the time when degeneracy-aware hybrid branching was included into SCIP indicated a 18.6 % speedup on the about 30 % of the instances in our MIP testset that are affected by the change.

4.4.2 Treemodel Scoring Rules

The treemodel method is a new way of scoring branching candidates based on their up and down (predicted) dual bound changes. Its theory and its SCIP implementation are based on the paper by Le Bodic and Nemhauser [54] and improved in the follow-up paper by Anderson, Le Bodic, and Morgan [8].

In SCIP, the default scoring function for branching candidates is the *product* score. Suppose that a variable x would result in a dual bound change of $\ell > 0$ on the “left” child and $r > 0$ on the “right” child, then the product rule gives a score

$$\mathit{product}(\ell, r) = \max(\ell, \epsilon) \cdot \max(r, \epsilon), \tag{13}$$

where $\epsilon > 0$ is a small constant. Experimentally, this produces smaller and balanced trees.

Le Bodic and Nemhauser [54] introduce two new scoring functions: the *Ratio* and the *Single Variable Tree Size* (SVTS) methods. Both are based on the abstract theoretical models developed in [54]. In a nutshell, Ratio corresponds to the growth rate of an abstract branch-and-bound tree, which would recursively branch on the same variable x and improve the dual bound of its children by ℓ and r, until all leaves reach or exceed a given gap value $G > 0$ (we simply refer to variable x as (l, r)). This is defined by the recurrence:

$$t(G) = \begin{cases} 1 & \text{if } G \leq 0, \\ 1 + t(G - l) + t(G - r) & \text{if } G \geq 0, \end{cases} \tag{14}$$

where $t(G)$ is the size of the tree required to close a gap G, starting from 0 at the root node, by repeatedly branching and improving the dual bound by l at the left child and r at the right child. In this setting the growth rate of a single-variable tree is the Ratio

$$\varphi = \lim_{G \to \infty} \frac{t(G+1)}{t(G)}.$$

For a large enough G, a variable (l, r) with the smallest Ratio φ requires the smallest single-variable tree. Therefore the Ratio scoring rule selects the variable with minimum Ratio φ among all candidates.

For the values l and r, we use the left and right dual bound changes predicted or computed by SCIP—these values could come from pseudocosts or strong branching. The Ratio φ is computed numerically using the methods given in [54], with some implementation improvements, namely a better parameter to choose the root-finding method and a caching of the previous fixed points for the fixed point method. The Ratio scoring function is particularly good for large trees, and can be used when the gap is infinite. It can be coupled with another scoring method to be used when the node being branched on is close to its leaves (according to a simple heuristic).

The second scoring function, SVTS, is based on the same abstract model, but is more complex than Ratio. Instead of characterizing the growth rate of an abstract tree when G is infinite, it takes a finite G as input and computes the size $t(G)$ of the abstract tree based on the single variable being evaluated for branching. The best variable according to SVTS is naturally the one which requires the smallest tree to close the gap G. The paper [54] gives a closed-form formula for SVTS, as well as an approximation that can be used when G is large, both of which are more efficient than using (14). One advantage of SVTS over Ratio is that it will not select variables that are good asymptotically when we are actually in the case where G is small and thus the expected size of the subtree is also small. If G is large, then SVTS will essentially behave like Ratio. When G is infinite, SVTS falls back to Ratio.

The implementation of the treemodel functions in SCIP is independent of any branching rule. Therefore, it could be employed by all branching rules that base their decision on the predicted dual bound improvements of the two child nodes, even though so far, only the hybrid branching rule of SCIP was adjusted to allow replacing the product score with the treemodel function.

The performance of SCIP with the product rule and the two treemodel functions on the MIPLIB 2017 benchmark set is given in Table 3. In the evaluation, instances solved in under one second, at the root node, or not solved by any scoring function, are ignored. Each instance is solved three times, with different random seed initializations.

Table 3 provides the number (resp. additional number) of instances solved by the product (resp. ratio and SVTS) scoring function. Relative differences compared to the product are given for time and number of nodes. We first consider all instances, then filter by number of nodes required by at least one scoring function, with thresholds at 10k, 50k and 100k. In this setup, SVTS is a clear winner, with a minor improvement over all instances (1% time), and a substantial improvement (11% time) for instances requiring at least 100k nodes for some setting.

Due to the relatively small improvement over all instances and the increased complexity of SVTS compared to product rule, the latter remains the default scoring function of SCIP. The treemodel rules can be turned on by the user (see "treemodel" setting files in the coverage directory), as they can be extremely beneficial on some instances, especially those with large trees.

Table 3: Performance results on instances of the MIPLIB 2017 benchmark set. From top to bottom: all instances, instances which require more than 10k, 50k and 100k nodes.

Results (set)	product			Ratio			SVTS		
	#	Time	Nodes	#	Time	Nodes	#	Time	Nodes
Total (all)	305	296.61k	135.63m	+1	1.04	0.88	**+3**	**0.96**	**0.82**
sh. geo. mean		351.50	**4.84k**		1.02	1.05		**0.99**	1.00
Total (≥10k)	141	216.35k	135.49m	+4	1.01	0.88	**+5**	**0.92**	**0.82**
sh. geo. mean		882.15	92.93k		0.99	1.04		**0.93**	**0.94**
Total (≥50k)	93	144.30k	134.87m	+4	1.00	0.87	**+5**	**0.86**	**0.81**
sh. geo. mean		1.03k	327.19k		0.96	1.02		**0.89**	**0.89**
Total (≥100k)	83	125.22k	134.50m	**+3**	1.03	0.87	**+3**	**0.89**	**0.81**
sh. geo. mean		1.09k	441.84k		0.96	1.01		**0.89**	**0.89**

4.4.3 Vanilla Full Strong Branching

Vanilla full strong branching is a textbook implementation of full strong branching for scientific experimentation purposes. It contrasts with SCIP's default full strong branching implementation, `fullstrong`, which includes additional features such as early variable domain reduction, early node cutoff, and propagation.

Here is a summary of the features specific to the `vanillafullstrong` branching rule, with respect to the features of the `fullstrong` branching rule:

- propagation is deactivated;
- node cutoff and variable domain reduction are deactivated. The `vanillafullstrong` branching rule can only result in `SCIP_BRANCHED` or `SCIP_DIDNOTRUN`;
- `idempotent` (optional, default FALSE): when activated, this feature leaves SCIP, as much as possible, in the same state before and after the branching call. Basically, `vanillafullstrong` will not update any SCIP statistic related to or resulting from strong branching computations;
- `donotbranch` (optional, default FALSE): when activated, no branching is done, and `vanillafullstrong` always results in `SCIP_DIDNOTRUN`. This allows for users to call the branching rule as an oracle only, that is, asking on which variable full strong branching would branch, without performing the actual branching step. The best branching candidate of the last branching call can then be retrieved by calling `SCIPgetVanillafullstrongData()`;
- `scoreall` (optional, default FALSE): when activated, `vanillafullstrong` will compute the strong branching score of all branching candidates, even if one of them results in node infeasibility (highest strong branching score possible). Otherwise, a candidate resulting in node infeasibility will cause `vanillafullstrong` to consider that candidate the best one for branching, and the remaining candidates not already processed will not be evaluated (and will have score `-SCIPinfinity`);
- `collectscores` (optional, default FALSE): when activated, `vanillafullstrong` will store the computed strong branching scores from the last call. The scores can then be retrieved by calling `SCIPgetVanillafullstrongData()`. Otherwise, `vanillafullstrong` will not store the scores, and `SCIPgetVanillafullstrongData()` will return a NULL pointer for the scores;
- `integralcands` (optional, default FALSE): when activated, `vanillafullstrong` will consider all non-fixed integer variables as candidates for branching, not only

those taking a fractional value in the current LP solution. More specifically, the branching candidates are obtained from `SCIPgetPseudoBranchCands()` instead of `SCIPgetLPBranchCands()`.

4.4.4 Updates to the Lookahead Branching Rule of SCIP

SCIP 7.0 comes with an updated version of the lookahead branching rule introduced in SCIP 6.0 [36]. First, new scoring schemes have been introduced to compute a score for a candidate variable based on the LP bounds predicted for the child nodes and potential grandchild nodes. The new default scheme computes the score as the product of the products of the dual bound gains for the two best pairs of grandchild nodes, with one pair per child node. This approach is similar to the product score (13). Additionally, the relative number of infeasible grandchild nodes is added, weighted with the average score obtained from any pairs of grandchild nodes. In computational experiments on the MMMC testset, the new scoring scheme reduced the average solving time, tree size, and fair node number [30] for instances solved within the time limit of two hours by more than 15 % compared to the old SCIP default.

Second, domain propagation was included into the lookahead branching scheme and grandchild node information is buffered and re-used if the same grandchild is evaluated again. A filtering mechanism was added that filters out unpromising candidates after the corresponding child nodes have been processed, but before grandchild node evaluation starts. Finally, pseudocost-based candidate selection in abbreviated lookahead branching uses pseudocosts (if already reliable) to select the most promising candidates for the lookahead evaluation.

The default setting for lookahead in SCIP 7.0 uses the abbreviated version and evaluates the best four candidates with lookahead branching, thereby evaluating the two most promising pairs of grandchild nodes per potential child node. In computational experiments on the MMMC testset, this version was able to reduce the average tree size by more than 20 %, but also leads to an increase in the average solving time by almost 20 %. Therefore, it is particularly suited for applications where tree size is a more important factor than sequential running time, for example, in a massive parallel environment with UG. Another prime application are machine-learning-based branching rules that aim at imitating a given reference rule. So far, the reference rule is typically full strong branching, but with just a bit more effort spent by lookahead branching in the offline training phase, the quality of the learned branching rule could be improved further. More details on the updated lookahead branching rule and computational results can be found in the PhD thesis of Gamrath [28].

4.4.5 Updated Branching Point Selection for External Candidates

The external branching candidates pool in SCIP allows constraint handlers to register variables as candidates for branching that are not necessarily integer variables with fractional value in the current relaxation solution. For example, the nonlinear constraint handlers use this feature to implement spatial branching.

If a constraint handler does not provide a value where to split the domain for an external branching candidate x_i, $i \in \mathcal{N}$, SCIP selects a branching point by taking the value of the variable in the current LP solution $\hat{x}$, if such solution was available. This point was moved further away from the variable bounds by projecting it onto $[\ell_i^{\mathsf{loc}} + \alpha(u_i^{\mathsf{loc}} - \ell_i^{\mathsf{loc}}), u_i^{\mathsf{loc}} - \alpha(u_i^{\mathsf{loc}} - \ell_i^{\mathsf{loc}})]$. Here, ℓ^{loc} and u^{loc} denote the lower and upper bounds, respectively, on the variables at the current node of the branch-and-bound tree. The value of α can be set via the parameter `branching/clamp` (default 0.2)

Motivated by Fischetti and Monaci [23], SCIP 7.0 moves the branching point for bounded variables even closer to the middle of the variable domain to ensure that the

current point is outside the domain for at least one child. Initially, the branching point is set to

$$\beta(\ell_i^{\mathsf{loc}} + u_i^{\mathsf{loc}})/2 + (1-\beta)\hat{x}_i$$

and then projected on $[\ell_i^{\mathsf{loc}} + \alpha(u_i^{\mathsf{loc}} - \ell_i^{\mathsf{loc}}), u_i^{\mathsf{loc}} - \alpha(u_i^{\mathsf{loc}} - \ell_i^{\mathsf{loc}})]$, where β can be set via the parameter `branching/midpull` and defaults to 0.75. If the local domain of a variable is small, compared to its global domain, then this shifting towards the middle of the domain is reduced. That is, if $(u_i^{\mathsf{loc}} - \ell_i^{\mathsf{loc}})/(u_i - \ell_i) < \gamma$, then β is replaced by $\beta(u_i^{\mathsf{loc}} - \ell_i^{\mathsf{loc}})/(u_i - \ell_i)$ in the previous formula. The value of γ can be set via the parameter `branching/midpullreldomtrig` and defaults to 0.5.

At the time this feature was added to SCIP, it lead to increasing the number of solved instances by 8 (1.4% of the MINLP testset) and reducing the mean solving time by 10% on those instances that were affected by this change.

4.5 Conflict Analysis

Conflict analysis, as it is implemented in SCIP and many other MIP solvers, relies on globally valid proofs of the form

$$\tilde{y}^\top A x \geq \tilde{y}^\top b, \tag{15}$$

where $\tilde{y} \in \mathbb{R}_+^m$. This type of infeasibility proof is used as a starting point for conflict graph analysis and dual proof analysis [77]. Usually, LP-based infeasibility with respect to the bounds $\ell \leq \ell' \leq u' \leq u$ is proven by a ray $(\tilde{y}, \tilde{r}) \in \mathbb{R}_+^m \times \mathbb{R}^n$ in the dual space of the LP relaxation

$$\max \left\{ y^\top b + \sum_{i \in \mathcal{N}:\, r_i > 0} r_i \ell_i' + \sum_{i \in \mathcal{N}:\, r_i < 0} r_i u_i' \mid y^\top A + r^\top = c^\top,\ y \in \mathbb{R}_+^m,\ r \in \mathbb{R}^n \right\}.$$

Farkas' lemma [22] states that if the relaxation of (1) is infeasible with respect to ℓ' and u', then there exist $(y, r) \in \mathbb{R}_+^m \times \mathbb{R}^n$ satisfying

$$\begin{aligned} y^\top A + r &= 0, \\ y^\top b + \sum_{i \in \mathcal{N}:\, r_i > 0} r_i \ell_i' + \sum_{i \in \mathcal{N}:\, r_i < 0} r_i u_i' &> 0. \end{aligned} \tag{16}$$

An infeasibility proof of form (15) is a direct consequence of (16) and is called *dual proof.* The dual proof is a globally valid constraint, since it is a non-negative aggregation of globally valid constraints. However, if locally valid constraints are present in the LP relaxation, e.g., local cuts, the resulting proof constraint might not be globally valid. In that case, conflict analysis is not directly applicable because so far conflict graph and dual proof analysis rely on globally valid proofs, see, e.g., [63, 1, 77]. The same situation appears when applying conflict analysis during LP-based spatial branch-and-bound for non-convex MINLPs. Here, linearization cuts are applied that might rely on local variable bounds. Consequently, these linearization cuts are only locally valid. In a computational study on general MINLPs, Witzig et al. [78] observed that for only 5% of all analyzed infeasible LP relaxations a globally valid infeasibility proof could be constructed. To "learn" from the remaining LP-based infeasibilities, *local dual proofs* are introduced with SCIP 7.0.

Let us distinguish between constraints needed for the definition of the model and additional constraints, that might be only locally valid, separated during the solving process. Without loss of generality, we assume that A represents all model constraints. For every subproblem s with local bounds ℓ' and u', let $\mathcal{G}^s$ be the index set of additional

constraints valid at s, $G^s \in \mathbb{R}^{p\times n}$ be the constraint matrix representing all those constraints and $d^s \in \mathbb{R}^p$ be the corresponding right-hand side. At every subproblem s, the LP relaxation reads

$$\min \{c^\top x \mid Ax \geq b,\ G^s x \geq d^s,\ \ell' \leq x \leq u'\}.$$

Further, we denote the index set of all additionally separated constraints that are globally valid by $\mathcal{G}^0$. During the branch-and-bound process, the set of additional constraints expands along each path of the tree, i.e., it holds that $\mathcal{G}^0 \subseteq \mathcal{G}^{s_1} \subseteq \ldots \subseteq \mathcal{G}^{s_p} \subseteq \mathcal{G}^s$ for every path $(0, s_1, \ldots, s_p, s)$. Consequently, the infeasibility proof of form (15) changes to

$$\tilde{y}^\top Ax + \tilde{w}^\top G^s x \geq \tilde{y}^\top b + \tilde{w}^\top d^s, \tag{17}$$

where $\tilde{w} \in \mathbb{R}^p_+$ denotes the vector of dual multipliers associated with all rows of G^s derived from a slight modification of (16). For a more detailed explanation we refer to [78].

In the previous releases of SCIP, only globally valid constraints were considered when constructing infeasibility proofs, i.e.,

$$\tilde{y}^\top Ax + \bar{w}^\top G^0 x \geq \tilde{y}^\top b + \bar{w}^\top d^0, \tag{18}$$

where $\bar{w}_j \coloneqq \tilde{w}_j$ if $j \in \mathcal{G}^0$ and $\bar{w}_j \coloneqq 0$ otherwise. The drawback of this approach is that (18) does not prove local infeasibility within the local bounds ℓ' and u' in general. In this case, the constraint of form (18) is not used for further considerations during conflict analysis. In other words, if locally valid constraints are needed to render LP-based infeasibility, conflict analysis was not applicable.

With SCIP 7.0, conflict analysis incorporates locally valid constraints to construct locally valid dual proofs of infeasibility of form

$$\tilde{y}^\top Ax + \hat{w}^\top G^s x \geq \tilde{y}^\top b + \hat{w}^\top d^s, \tag{19}$$

incorporating all local constraints represented in the index set $\hat{\mathcal{G}}$, with $\mathcal{G}^0 \subseteq \hat{\mathcal{G}} \subseteq \mathcal{G}^{s_p}$, where $\hat{w}_j \coloneqq w_j$ if $j \in \hat{\mathcal{G}}$ and $\hat{w}_j \coloneqq 0$ otherwise. The infeasibility proof (19) is valid for the search tree induced by subproblem q with

$$q \coloneqq \operatorname*{argmin}_{q\in\{0,s_1,\ldots,s\}} \left\{\mathcal{G}^{q-1} \subseteq \hat{\mathcal{G}},\ \hat{G} \cap (\mathcal{G}^{q-1} \setminus \mathcal{G}^q) = \emptyset\right\}.$$

Hence, the infeasibility proof might be lifted to an ancestor q of the subproblem that it was created for, if all local information used for the proof was already available at subproblem q. In SCIP 7.0, local infeasibility proofs are only used for dual proof analysis, and not for conflict graph analysis. However, it would be possible to apply conflict graph analysis to (19). Unfortunately, this would introduce a computational overhead because the order of locally applied bound changes and separated local constraints needs to be tracked and maintained. Since conflict graph analysis already comes with some overhead due to maintaining the so-called delta-tree, i.e., complete information about bound deductions and its reasons within the tree, we omit applying conflict graph analysis on locally valid infeasibility proofs.

When this feature was contributed to SCIP, we did not observe a performance impact on our internal MIP test set over 5 random seeds. However, 9 additional instances could be solved. On our internal MINLP test set, we did observe a performance improvement of 4.1% on affected instances over 5 permutations.

4.6 Tree Size Estimation and Clairvoyant Restarts

Traditionally, SCIP reports the quality of the current incumbent solution in terms of the (primal-dual) gap. While the gap provides a guarantee on the maximum percentage deviation from the optimal value, its use as a measure of the MINLP search progress is very limited. We expect that many users of SCIP have experienced situations where the gap was already closed to 99 % after a few seconds, but most of the runtime is spent on closing the remaining 1 %. With version 7.0, SCIP has been extended by several methods that can be used instead of the gap as a more accurate "progress bar" to approximate the remaining time of the search. In this section, we only give a high-level explanation of these methods to help the user understand the extended SCIP output. We refer the interested reader to the technical report by Hendel et al. [40], which explains the underlying concepts in greater detail.

In total, we have four atomic measures of the *search completion*, by which we denote the fraction of already solved nodes of the final search tree. Besides the current amount of closed gap, which lies between 0 and 1, we implemented the Sum of Subtree Gaps (SSG) [67], which aggregates the gaps of different local subtrees into a monotone measure that reaches 0 when the search terminates. The other two measures of search completion are the tree weight and the leaf frequency. The tree weight accumulates the weights of leaf nodes (nodes that were (in-)feasible or pruned by bound) of the search tree, where a leaf at depth d weighs 2^{-d}. The tree weight is monotone and equals 1 at the end of the search in the default case of binary branching decisions. The leaf frequency measures the ratio of leaf nodes among all nodes, a quantity that reaches $\frac{1}{2}$ when the search ends.

Experimental results confirm that all three measures—SSG, tree weight, and leaf frequency—constitute substantially better approximations of search completion than the (closed) gap, and therefore better estimations of the final tree size. The search completion approximations consitute the first group of techniques to estimate tree size. Another group of techniques considers the above measures as time series and applies forecasting techniques such as double-exponential smoothing [43], which also takes into account changes in the trend of a measure. Initially, each leaf node of the search tree corresponds to one time series step. Later, we aggregate $k > 1$ leaf nodes into a single time series step. We increase k, which we call the resolution, in powers of 2 to adapt the time series representation to larger search trees.

As third and final group of techniques, we provide two models that combine features from several of the above time series. The first model computes a linear combination of the tree weight and SSG. This linear combination is again a measure of search completion and can be calibrated to specific problem instances to be more accurate than the two individual measures by themselves. Since both SSG and tree weight are monotone, we call this method monotone linear regression. By default, SCIP outputs the current value of this monotone linear regression alongside gap in a new display column for search completion.

The second model uses both values and time series trends of all four measures, as well as the trend in the open nodes, to approximate search completion via random forest regression [15]. In our experiments, a suitably trained random forest outperforms all other tested methods by a considerable margin. It cannot be expected that there is a single regression forest that works well for all possible input instances. Therefore, SCIP 7.0 can read and use regression forests that are provided by the user in an extended `CSV`-format via the new string parameter `estimation/regforestfilename`. We also provide an `R`-script to train a regression forest on user data and output the forest in the required format, see the SCIP documentation for instructions.

Table 4: Meaning of values for parameter `misc/usesymmetry`.

value	meaning
0	no symmetry handling
1	for each component, use orbitopes if applicable, otherwise use symresacks
2	orbital fixing for each component
3	for each component, use orbitopes if applicable, otherwise use orbital fixing (default)

4.7 Clairvoyant Restarts

In versions prior to 7.0, SCIP restarted the search only if a certain fraction of integer variables could be fixed during the root node. In SCIP 7.0, we now enable in-tree restarts based on tree size estimation, which we call clairvoyant restarts [7]. In a nutshell, the search process is restarted from the root node if, after a suitable initialization of the search tree when at least 1k leaf nodes have been explored, the estimated search completion is less than 2 %, based on a forecast of the tree weight time series By default, at most one clairvoyant restart is performed.

Enabling clairvoyant restarts is responsible for a 10 % speed-up on the harder instances of our internal MIP test set, and 2 % overall. Interestingly, clairvoyant restarts result on average in a smaller number of search nodes, even if the two trees before and after the restart are combined. We explain this by the fact that the search information collected during the first tree search can be reused to make more informed branching decisions. This in turn helps to reduce the size of the second, final search tree. Collected search information that is reused after a restart includes primal solutions, cutting planes, conflict constraints and variable branching history, such as pseudo costs,.

4.8 Improvements in Symmetry Handling

Symmetries in mixed-integer programs typically have an adverse effect on the running time of branch-and-bound procedures. This is typically due to symmetric parts of the branch-and-bound tree being explored repeatedly without providing new information to the solver.

Since computing the full symmetry group of an integer program is $\mathcal{NP}$-hard, see Margot [62], SCIP only computes the symmetry subgroup that keeps the problem formulation invariant. Define the image of a permutation γ of an n-dimensional vector x to be $\gamma(x) := (x_{\gamma^{-1}(1)}, \dots, x_{\gamma^{-1}(n)})$. The *formulation symmetry group* of an integer program $\min\{c^\top x : Ax = b,\ x \in \mathbb{Z}^n\}$ with m constraints is the largest permutation group that satisfies for every $\gamma \in \Gamma$:

- $\gamma(c) = c$ and
- there exists a permutation π of the rows of A such that $\pi(b) = b$ and $A_{\pi(i)\gamma(j)} = A_{ij}$.

In the following, we assume that Γ is the (formulation) symmetry group, which is currently computed within SCIP using the graph isomorphism package `bliss` [46]; see [68] for details of the approach.

To handle symmetries on binary variables, two symmetry handling approaches have been implemented and are available in SCIP since version 5.0: a pure propagation approach, *orbital fixing* (OF) [61, 65, 66], and a separation-based approach via *symretopes* [42]. The latter approach is implemented using three constraint handler plugins that take care of different kinds of symretopes: orbitopes [48], orbisacks [47, 56], and symresacks [41, 42]. In both approaches, the user has the possibility to use the symmetries of the original or the presolved problem as the basis for symmetry reductions.

Symmetry is controlled via the `misc/usesymmetry` parameter; its meaning is described in Table 4.

The SCIP 7.0 implementation of computing and handling symmetries is relatively involved. We therefore provide a flow diagram for the computation of symmetries in Figure 3. The described mechanism is triggered during the presolve phase if symretopes shall be added and whenever orbital fixing is called.

General Enhancements With the release of SCIP 7.0, the symmetry code has been completely restructured. In the previous releases, the presolver plugins `presol_symmetry` and `presol_symbreak` stored symmetry information and added symretope constraints, respectively. Moreover, the propagator plugin `prop_orbitalfixing` was responsible for running orbital fixing. In the current release, this functionality has been unified in the new propagator plugin `prop_symmetry`, avoiding the storage of the same symmetry information in different plugins. General symmetry related methods like orbit computations are implemented in the new file `symmetry.c`.

Besides refactoring the code structure, new data structures have been introduced to enhance the implementation. If the symmetry group is generated by k permutations of n variables, the previous release stored this information in a $k \times n$ matrix. In SCIP 7.0, also the transposed $n \times k$ matrix is stored to perform, for example, orbit computations more (cache) efficiently.

Another feature of SCIP 7.0 is to check whether the (formulation) symmetry group Γ is a product group $\Gamma = \Gamma_1 \otimes \cdots \otimes \Gamma_k$ in which the factors Γ_i act independently on pairwise distinct variables. We call the factors *components* of the symmetry group. By taking components into account, again orbit computations and other symmetry related methods can be performed more efficiently. But the main advantage is that different symmetry handling methods can be used on the different components. For example, it is possible to use orbital fixing for one component and symretopes for another. To implement this, the symmetry control parameter `misc/usesymmetry` has been extended. It is an integer parameter with range $\{0, \dots, 3\}$ encoding a bitset: the 1-bit encodes whether symretopes are active; the 2-bit encodes if orbital fixing is enabled, see Table 4 for a description. The exact behavior is as follows: If both symretopes and orbital fixing are activated, SCIP 7.0 will try to use orbitopes if available, and otherwise will fall back to orbital fixing. If only symretopes are activated, then SCIP 7.0 will try orbitopes first, and fall back to symresacks. Note that symretope constraints currently can be only added during presolving. That is, if the symretopes are selected, symmetries are computed at the end of presolving (the current default). If only orbital fixing is selected, symmetries are computed when the branching process starts (the default in the previous SCIP release).

Since memory consumption for storing the permutations matrices can be large for large instances, SCIP 7.0 allows the compression of symmetry information. If compression is enabled via the parameter `propagating/symmetry/compresssymmetries`, all variables that are not affected by the symmetry group are removed from the permutation matrices if the percentage of affected variables is below a threshold. The threshold can be adapted via the parameter `propagating/symmetry/compressthreshold`.

To speed up symmetry detection, SCIP 7.0 checks whether all objective coefficients in the problem formulation are different. In this case, no formulation symmetry can be present, because no non-trivial permutation can fix the objective coefficient vector, compare the second paragraph of this section. Moreover, since SCIP is currently only able to handle symmetries of binary variables, SCIP computes symmetries and checks whether a binary variable is affected by the computed symmetry group. If this is not the case, all symmetry handling methods are disabled. Otherwise, if symmetries have been detected, SCIP forbids multi-aggregation of affected binary variables. The reason for this is that the symmetry handling methods may fix binary variables, and transferring

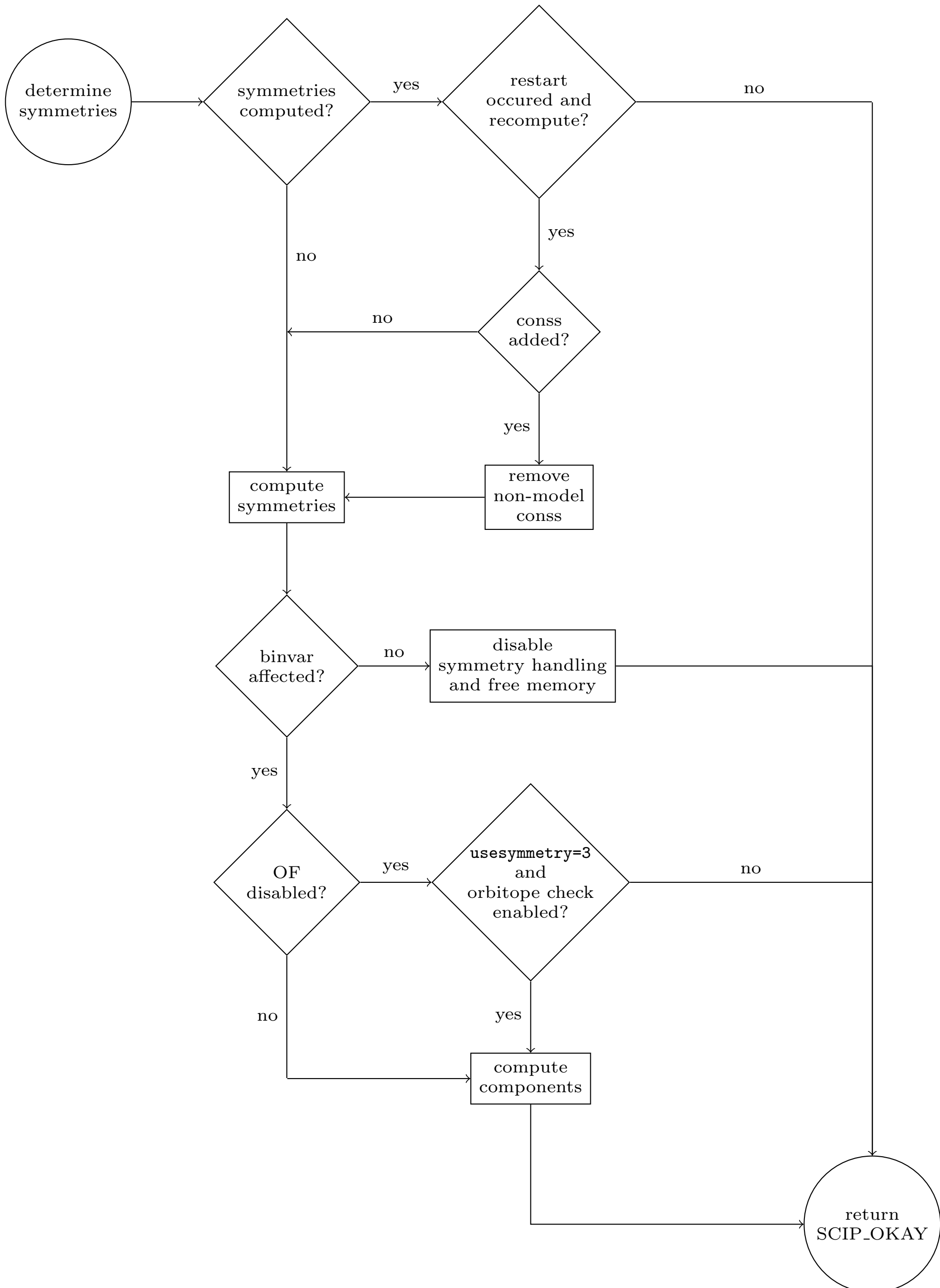


Figure 3: Flow diagram for computing symmetries.

the fixing of a multi-aggregated variable to the active variables in the multi-aggregation might not be possible.

To increase the number of instances in which symmetries can be detected, the symmetry detection routine of SCIP 7.0 can also deal with certain bound disjunction constraints. Further, the routine can negate equations to be able to detect symmetric equations with flipped signs. Finally, with the release of SCIP 7.0 it is possible to recompute symmetries after a restart, even if symretope constraints have been added to handle symmetries.

Enhancements Regarding Symretopes To be able to recompute symmetries in the presence of symretope constraints, these constraints have been endowed with a flag encoding whether they are model constraints, i.e., present in the original formulation or have been added to handle symmetries. In the latter case, these constraints can be removed from the problem which allows the symmetry detection routine to recompute the full set of symmetries.

In contrast to orbisack and symresack constraints, an orbitope constraint is not defined by a single permutation. Instead it requires the presence of a symmetry group with a certain structure, i.e., a full symmetric group acting on the columns (or rows) of a 0/1 matrix. For each component of the symmetry group, SCIP 7.0 checks whether the group restricted to the component has the required structure. If symretopes are enabled, the corresponding orbitope constraint is added to the problem. Since stronger symmetry handling inequalities are available if, additional to the correct group structure, certain set packing or partitioning constraints are present, SCIP also checks for the presence of these constraints. If the necessary set packing or partitioning constraints are present, orbitopes are upgraded to packing and partitioning orbitopes. While a detection mechanism for the required set packing or partitioning structures was already contained in previous releases, a more efficient implementation is available in SCIP 7.0.

If orbitopes cannot be upgraded to packing or partitioning orbitopes, the orbitope is a full orbitope. For full orbitopes, Bendotti et al. [11] developed a complete propagation algorithm, that is, an algorithm that detects all symmetry-based variable fixings that can be deduced at a node of the branch-and-bound tree. Their algorithm can be implemented in a static way or in a dynamic fashion. The static version uses the natural variable order for symmetry reductions, whereas the dynamic version adapts the variable order to the branching decisions made from the root node to the current node. In SCIP 7.0, an implementations of the static version and a modified dynamic version are available. The modified version uses a global variable order that is based on branching decisions instead of an adapted order for each node of the branch-and-bound tree. The global ordering allows the use of more efficient data structures. The idea for this variable order is the same as the global rank used by Margot [61]. The trade-off is that the full potential of the dynamic version cannot be exploited.

Similar to orbitopes, symresacks can be upgraded to packing and partitioning symresacks if certain packing/partitioning constraints are present, see [41]. While this feature was only available for certain permutations in previous releases, the user can enable this upgrade for arbitrary permutations in SCIP 7.0. By default, this feature is disabled because the data structures for general permutations are less efficient and only specially structured permutations seem to appear in practice.

Enhancements Regarding Orbital Fixing Orbital fixing is a propagation method that deduces variable fixings at a node of the branch-and-bound tree based on the branching decisions that have been carried out from the root node to the current node. To ensure correctness of orbital fixing, all variable fixings that are deduced by other components of SCIP have to be compatible with orbital fixing. This is guaranteed if the components implement *strict setting algorithms*, i.e., if a variable x is fixed to some value a, then the

components are (in principle) able to also fix $\gamma(x) = a$ for each symmetry $\gamma \in \Gamma$, see Margot [61]. Previous releases of SCIP ensured correctness of orbital fixing by disabling plugins that might interfere with orbital fixing. Since fixings can only be in conflict with orbital fixing if they have been performed after symmetry has been computed, SCIP 7.0 now stores all binary variables that have been fixed globally and takes these variables into account while performing orbital fixings. Other components of SCIP thus do not have to be disabled if orbital fixing is active.

4.9 Updates to the Generic Benders' decomposition Framework

SCIP 7.0 greatly extends the functionality of the Benders' decomposition (BD) framework and improves its computational performance. Two major features have been introduced to the framework—parallelization and the ability to handle convex MINLP subproblems. Additionally, a number of key enhancement techniques have been implemented, such as cut strengthening and new primal heuristics. Finally, additional parameter settings have been introduced to provide more control over the features of the BD framework. For a detailed overview of the BD framework and a number of the features highlighted here, the reader is referred to Maher [58].

Many of the features and enhancements that are described here are implemented directly in the BD core of SCIP. As such, they are available to all BD plugins that are developed for SCIP, such as the `default` and `gcg` plugins. The runtime parameters associated with the features and enhancements can be controlled for each BD plugin, thus the parameters are identified by the BD plugin name. In the following, `<X>` will be used to represent `benders/<benderspluginname>`.

4.9.1 Features

Parallelization There is a long history of employing parallelization techniques to reduce the wall clock time when applying BD [55, 59]. This typically stems from the fact that the application of BD exposes a set of disjoint subproblems that can be solved concurrently. In the traditional implementation of BD [55, 59], parallelization can be employed to accelerate the solving process of the master problem and subproblems independently. In this case, parallel branch-and-bound can be used to solve the master MIP, then given a master solution the disjoint subproblems can be solved concurrently. In modern branch-and-cut implementations [58], these two uses of parallelization must be integrated. The current release of the SCIP Optimization Suite, to the best of our knowledge, is the first Benders' decomposition framework to combine master problem and subproblem parallelization.

The traditional parallelization of BD, solving the disjoint subproblems concurrently, is executed during constraint enforcement and solution checking. OpenMP [16] is used to parallelize the subproblem solving process. As explained in Maher [58], all of the subproblems are solved prior to the generation of Benders' cuts. The cut generation, including addition to the master problem, then proceeds in a set subproblem order, independent of the solving time for the individual subproblems. Thus, the solution algorithm will remain deterministic even with the use of parallelization. After building SCIP with the option `-DTPI=omp`, the parameter `<X>/numthreads` will set the number of threads available for solving the BD subproblems in parallel.

An important aspect of parallel algorithms is ensuring an equal amount of work is distributed to all processes—known as load balancing. To support load balancing when solving the BD subproblems in parallel, a priority queue has been implemented to define the solving order. If all subproblems are solved in each call to the BD constraint handlers, then *harder* subproblems—identified as having a higher average number of

LP/NLP iterations—will take a higher priority. Otherwise, load balancing is ignored and the subproblem that has been called the least often is prioritized to ensure that all subproblems are called regularly. If not all subproblems are solved in each call to the BD constraint handlers, but they have been solved the same number of times, then the *hardness* metric described above is used to form the priority order. The tie breaker in all cases is the subproblem index, where the lower index will be prioritized. This comparison of subproblems can be modified in custom BD implementations by providing a comparator to the function `SCIPsetBendersSubproblemComp`.

The parallelization of the branch-and-cut algorithm when employing BD has been achieved primarily by extensions to UG. Within SCIP, the transfer of cuts between SCIP instances, which was previously only used for the Large Neighborhood Benders' Search [57], has been improved. The function `SCIPstoreBendersCut` can be called from Benders' cut plugins to store the constraint information for the corresponding cut, specifically variables, coefficients, left-hand side and right-hand side. The stored cuts can be retrieved by calling `SCIPbendersGetStoredCutData`. Applying all stored cuts to a given SCIP instance is achieved using the function `SCIPapplyBendersStoredCuts`. In order to use the branch-and-cut parallelization of UG with BD, the Benders' cuts must be representable as linear constraints and stored in the Benders' cut plugins when they are created. The extensions to UG that enable the parallelization of the branch-and-cut approach for BD will be described in Section 6.

Improved Handling of MINLPs The BD framework is designed to handle general CIP subproblems; however, in the previous release convex Benders' cuts could only be generated from LP subproblems. SCIP 7.0 extends the BD framework to enable the generation of Benders' cuts from the NLP relaxation of convex MINLPs. Primarily, this involved extending the default subproblem solving methods from the BD core to solve NLP relaxations of the subproblem if they exist. Additionally, the classical optimality and feasibility Benders' cuts—the `benderscut_opt` and `benderscut_feas` plugins—include methods for generating cuts from the result from solving the NLP relaxation.

A convexity check has been added to identify whether an MINLP or NLP contains only convex constraints. Currently, an MINLP or NLP is identified as convex if it is comprised only of linear constraints and nonlinear constraints of type `abspower`, `nonlinear`, and `quadratic` and these constraints are convex. If the MINLP or NLP contains any other constraints, then it is identified as non-convex and thus classical optimality and feasibility Benders' cuts will not be generated.

Finally, an additional feasibility Benders' cut plugin has been implemented to support convex MINLP subproblems. Consider a Benders' subproblem of the form,

$$z(\hat{x}) = \min\{d^\top y \,:\, g(y) \geq h - T\hat{x},\ y \in \mathbb{R}^n_+\}, \tag{20}$$

where g is a convex nonlinear function and $\hat{x}$ is a master problem solution provided as input. If the subproblem is infeasible, an auxiliary subproblem, given by

$$z'(\hat{x}) = \min\{\mathbb{1}^\top \nu \,:\, g(y) + \nu \geq h - T\hat{x},\ y \in \mathbb{R}^n_+,\ \nu \in \mathbb{R}^m_+\}, \tag{21}$$

is solved to generate a feasibility cut. The auxiliary subproblem $z'(\hat{x})$ minimizes the violation of the constraints in the infeasible problem. The dual solution from this auxiliary problem can be used to generate a feasibility cut using the methods provided in `benderscut_opt`.

Apply Benders' decomposition using Supplied DEC File Section 4.2 presents the new feature for handling used-defined decompositions within SCIP. Upon supplying a used-defined decomposition, it is possible to directly perform an automatic reformulation and apply BD. Importantly, the parameter `decomposition/benderslabels` must be

set to `TRUE` to ensure that the appropriate variable labels are assigned. Then setting `decomposition/applybenders` to `TRUE` will inform SCIP that a Benders' decomposition must be applied using the highest priority decomposition.

Applying BD using a decomposition in the DEC format will invoke the default BD plugin and cut generation methods. However, it is still possible to customize the BD algorithm. This can be achieved by designing and including custom BD cut generation methods as part of a user-defined SCIP solver.

Generate Optimality Cuts from External Subproblem Solutions In the previous release, the Benders' cut plugin for the classical optimality cut required the solution of a SCIP instance to compute the relevant linear constraint. This limited the flexibility of the BD framework, since the default Benders' cut plugin could not be used with custom subproblem solving methods. The API of the optimality Benders' cut plugin has been modified to enable the generation of optimality cuts from external subproblem solutions.

4.9.2 Algorithmic Enhancements

Cut Strengthening A simple in-out cut strengthening approach has been implemented to accelerate the convergence of the BD algorithm. The approach follows the method described in Fischetti et al. [24].

Given some core point x^o and the current LP relaxation solution x, the separation point that is provided to the BD subproblems to generate cuts is $x_{\text{SEP}} = \lambda x^o + (1-\lambda)x$, where $0 \leq \lambda \leq 1$. If, after k calls to the BD subproblems for constraint enforcement, no improvement in the lower bound is observed, then the separation point is set to $x_{\text{SEP}} = x + \delta$, where the entries of δ are chosen uniformly at random from $[0, \epsilon)$ with $\epsilon \ll 1$. If, after a further k calls to the BD subproblems for constraint enforcement, no improvement in the lower bound is observed, then the separation point is set to $x_{\text{SEP}} = x$. Also, after each call to the BD subproblems for constraint enforcement the core point is updated as $x^o \leftarrow \lambda x^o + (1-\lambda)x$.

The values of λ, ϵ and k are set by runtime parameters. The respective parameters are `<X>/cutstrengthenmult`, `<X>/corepointperturb` and `<X>/noimprovelimit`, with default values of 0.5, 10^{-6} and 5.

A number of initialization solutions are provided for the core point x^o. The possible initialization solutions are i) the first LP relaxation solution, ii) the first primal feasible solution, iii) a solution vector of all zeros, iv) a solution vector of all ones, v) a relative interior point for the master problem and vi) the first primal feasible solution but reset to each new incumbent solution. The run time parameter `<X>/cutstrengtheninitpoint` is used to set the initialization point and is v) by default. The impact of the different initialization points on the effectiveness of cut strengthening is examined in Maher [58].

Trust Region Heuristic Trust region methods are applied within the BD algorithm to reduce the distance between the master problem solutions that are passed to the subproblems in consecutive calls. Limiting the distance between consecutive master solution has been shown to improve the convergence of the BD algorithm [55, 70, 69]. While many previous approaches applied a trust region directly to the master problem, this is not possible for the BD framework in SCIP since it requires a modification to the master problem's objective function.

Since the traditional usage of trust regions is not possible in SCIP, this concept was used to devise a BD-specific primal heuristic. The full details of the trust region heuristic can be found in Maher [57].

Feasibility Phase The application of BD involves the partition of problem constraints between a master problem and a set of subproblems. This partition may assign structural constraints to the subproblems that are necessary to ensure feasibility for the original problem. As a result, master problem solutions $\hat{x}$ induce infeasible instances of the subproblems, requiring the generation of feasibility cuts.

Classical feasibility cuts are well known to be weak, requiring a large number to escape a region of the master problem that induces infeasible subproblems. Also, the strengthening technique described above cannot be used to improve the quality of feasibility cuts. To address this limitation of feasibility cuts, a feasibility phase has been developed for the BD framework.

The feasibility phase is modeled after the phase 1 of the simplex method. This approach is only executed while solving the root node of the master problem. Hence, this requires the three-phase method to be enabled [58], which is achieved by setting `constraints/benderslp/active` to `TRUE`.

The feasibility phase replaces (20) with

$$z''(\hat{x}) = \min \{d^\top y + M\mathbb{1}^\top \nu \, : \, g(y) + \nu \geq h - T\hat{x}, \; y \in \mathbb{R}^n_+, \; \nu \in \mathbb{R}^m_+\}, \tag{22}$$

where M is a positive weight, initially set to a small value, to penalize the violation of constraints. Importantly, (22) is guaranteed to always be feasible. The feasibility property of this subproblem is important, since it enables the generation of classical optimality cuts and the use of cut strengthening methods. While (22) is similar to (21), the inclusion of $d^\top y$ in the objective function ensures that the optimality cuts generated from (22) provide a valid underestimator for the subproblem objective value.

The default BD algorithm is employed to solve the LP relaxation of the master problem in the root node. Thus, valid BD optimality cuts are generated by solving (22) as the BD subproblem. If the master problem LP relaxation supplies $\hat{x}$ to the subproblems for which no optimality cut can be generated, but there is a non-zero value in the solution vector ν, then M is set to $10M$. Following the modification of the subproblem objective function, the BD algorithm is restarted. This process continues until no optimality cuts can be generated for $\hat{x}$ and the solution vector $\nu = 0$.

The parameters `<X>/execfeasphase` and `<X>/slackvarcoeff` are used, respectively, to enable the feasibility phase and set the initial value of M, which is 10^6 by default.

Enabling Presolving and Propagation with Benders' decomposition The handling of the master problem variables has been improved allowing the use of SCIPs presolving and propagation methods. Specifically, within the Benders' decomposition constraint handler (`cons_benders`), an up and down lock is added for every master problem variable. For every auxiliary variable a down lock is added. The addition of the locks ensures that no invalid reductions are performed on the master problem.

4.9.3 New Parameters for Benders' decomposition

A number of new parameters have been added to control the behavior of the BD algorithm within SCIP.

Implicit Integer Auxiliary Variables If the BD subproblem is defined as a CIP, then it is possible that the objective function is always integral. In this situation, this means that the auxiliary variables are also always integer in the optimal solution. The parameter `<X>/auxvarsimplint` can be set to `TRUE` so that if the subproblem objective is always integral then the auxiliary variables are set as implicit integer variables. By default, this parameter is set to `FALSE`.

Cutting on All Solutions For the completeness of the BD algorithm when using the branch-and-cut approach, Benders' cuts need only be generated during the enforcement of constraints. While the BD subproblems are solved during the checking of primal solutions, cuts need not to be generated. However, cuts generated while checking solutions feasible for the master problem may be stronger, as these solutions are more likely to be in the interior of the master feasible region.

The parameter `<X>/cutcheck` can be set to `TRUE` so that Benders' cuts are also generated while checking primal solutions. The effectiveness of generating Benders' cuts while checking primal solutions is examined by Maher [58].

Improved Handling of the Large Neighborhood Benders' Search The Large Neighborhood Benders' Search was introduced in the previous release and described in more detail by Maher [57]. Since the last release, two additional parameters have been introduced to limit the number of calls to the Benders' decomposition subproblems while solving the auxiliary problem of large neighborhood search heuristics. The parameter `<X>/lnsmaxcallsroot` will limit the number of calls to the BD subproblems while solving the root node of the auxiliary problem. Similarly, `<X>/lnsmaxcalls` will limit the total number of calls to the BD subproblems.

Three-phase Method The three-phase method is a classical technique that involves solving the root node relaxation of the master problem by BD. Since the three-phase method is implemented using a constraint handler in SCIP, it is also possible to apply BD to solve relaxations at other nodes throughout the branch-and-cut tree. This is facilitated through the introduction of four parameters:

- `constraints/benderslp/maxdepth`: the maximum depth at which BD is used to solve the node relaxation. Default is 0 (only the root node), while -1 means all nodes and `maxdepth` > 0 means all nodes up to a depth of `maxdepth`.
- `constraints/benderslp/depthfreq`: if `maxdepth` > 0 and `depthfreq` $= n > 0$, then after `maxdepth` is reached, BD is used to solve the relaxation at nodes of depth $d = n, 2n, 3n, \ldots$ If `depthfreq` $= 0$, then BD is only used to solve the node relaxation up to `maxdepth`.
- `constraints/benderslp/stalllimit`: if `stalllimit` > 0, then BD will be used to solve the relaxation of the next processed node after n nodes are processed without a lower bound improvement.
- `constraints/benderslp/iterlimit`: at most `iterlimit` relaxation solutions at each node will be used to generate BD cuts, regardless of the other parameter settings.

4.9.4 Miscellaneous Changes

- When calling `SCIPcreateBendersSubproblem`, no SCIP pointer for the subproblems is needed anymore. It is possible to supply a `NULL` pointer if the subproblem is to be solved by an alternative algorithm, such as a shortest path algorithm.
- The handling of subproblem solving results has been improved. These have been standardized to `SCIP_RESULT`. The result from solving the convex relaxation and the CIP will use the `SCIP_RESULT` codes.
- The `SCIPcopyBenders` and the copy callback in the Benders' decomposition plugins now have a parameter `threadsafe`. If `threadsafe` is set to `TRUE`, then the actions performed during the copy and the resulting copy must be thread safe. More details regarding the use of the `threadsafe` parameter will be given in Section 6.

4.10 Technical Improvements and Interfaces

A set of smaller technical changes and improvements have been implemented with SCIP 7.0.

Numerical Emphasis Setting SCIP 7.0 comes with a new parameter emphasis setting "`numerics`" for numerically difficult problems that require safer numerical operations. In rare cases, such as SCIP solutions that are not feasible for the original problem after retransformation, the reason often lies in repeated (multi-)aggregations of variables. Because of SCIP's constraint-based design, the numerical consequences of multi-aggregations are not always visible to the responsible presolver. The numerical emphasis setting restricts the maximum dynamism in multi-aggregations and modifies or disables some other numerically critical operations, mostly in presolving. Note that this setting may slightly deteriorate performance. In troublesome cases where this emphasis setting does not resolve encountered infeasibilities in the original space, it is recommended to tighten the feasibility tolerance and/or completely disable multi-aggregations.

Relaxation-only Variables To formulate strong relaxations, it can be beneficial to add new variables that are only used to tighten a relaxation but do not appear in any CIP constraint. Within SCIP, this could lead to wrong results, for example when a copy of such a variable was fixed in a sub-SCIP (because no constraint was locking the variable) and this fixing was transferred back to the main SCIP instance. In SCIP 7.0, it is now possible to mark a variable with zero objective coefficient as "relaxation-only". As relaxation-only variables must not appear in any constraint, they can be and are excluded by the problem copying methods. Thus, if constraint compression is also disabled, sub-SCIPs can now potentially have less variables than the main SCIP.

Extended Constraint Upgrades The structure recognition in nonlinear constraints has been slightly extended. Expressions of the form $|x|^p x$ in a general nonlinear constraint are now replaced by a new variable-constraint pair z and $z = |x|^p x$, where the new constraint is handled by the `abspower` constraint handler.

When checking whether a quadratic constraint can be reformulated as a second-order-cone (SOC) constraint, binary variables are now handled as if they were squared, as this might allow for more SOC reformulations. For example, $x_i^2 + x_j \leq x_k^2$ for some i, $k \in \mathcal{N}$ and $j \in \mathcal{I}$ with $\ell_j = 0$, $u_j = 1$, and $\ell_k \geq 0$ is reformulated to the SOC constraint $\sqrt{x_i^2 + x_j^2} \leq x_k$.

LP Solution Enforcement by Tightening the LP Feasibility Tolerance Constraint handlers may struggle to enforce the solution of a relaxation if the relaxation has not been solved to a sufficient accuracy. For a (simplified) example, consider a constraint $x^5 = 32$ and let x be fixed to 2 in the current node of the branch-and-bound tree. As the relaxation solver is allowed to compute solutions that violate variable bounds up to a certain tolerance, it may return a solution $\hat{x}$ with value $\hat{x} = 2.000001$. As $2.000001^5 = 32.00008\ldots$, the constraint handler for $x^5 = 32$ cannot claim feasibility with respect to the default absolute feasibility tolerance of 10^{-6}. However, as the variable is already fixed, it also has no way to enforce the constraint by a bound tightening or branching. Thus, for LP relaxations, SCIP 7.0 now allows to (temporarily) reduce the primal feasibility tolerance of the LP solver and request a resolve of the LP (by returning `SCIP_SOLVELP` as result in a constraint handler's `ENFOLP` callback). The LP primal feasibility tolerance is reset to its original value when the node has been solved.

Raspbian Installers SCIP 7.0 has been compiled and tested on Raspberry Pi machines and Raspbian installers are available for download.

Glop LPI SCIP 7.0 now contains an interface to the LP solver Glop, which is part of the `Google OR-Tools`[1]. SCIP works quite reliably together with Glop, although it is currently slightly slower than other open source solvers.

Support for Continuous Linking Variables in Linking Constraint Handler A linking constraint in SCIP is represented by two equations

$$y = \sum_i a_i x_i, \quad \sum_i x_i = 1, \quad x_i \in \{0, 1\},$$

to model the domain of the linking variable y. The linking variable and the coefficients previously had to be of integer type. The linking constraint handler has been refactored to additionally allow for linking variables of continuous type.

Calculation of Dual Integral During Solving Besides the computation of the primal-dual integral that is updated at each change of the primal or dual bound during the search, SCIP now also computes a primal and a dual reference integral with respect to a user-specified reference objective value. The computed values correspond to the primal integral [12] and its dual counterpart if the reference value is the known optimal solution value of an instance. Note that the reference objective value is only used for statistics and exploited at no time during the solving process.

Reformulation of Binary Products The quadratic constraint handler reformulates products of binary variables during presolving by introducing additional variables and constraints. Until the release of SCIP 7.0, the constraint handler avoided using AND constraints for reformulating these products. For SCIP 7.0, this behavior changes by adjusting the default value of the parameter `constraints/quadratic/empathy4and`, which forces the constraint handler to use AND constraints whenever it is possible.

5 SoPlex

SoPlex 5.0 is a major update on the last version, incorporating many technical changes.

5.1 Multiprecision Support

In previous versions, the arithmetic precision of the simplex routines had to be fixed at compile time and could only be selected from float, double and long double precision. With SoPlex 5.0, it has become possible to set an arbitrary arithmetic precision at runtime. For this purpose, SoPlex uses the Boost multiprecision library [14], which wraps either MPFR [25] (if available), or uses its own implementation of multi-precision floating-point numbers. The precision can be set by running SoPlex with parameters `solvemode` and `precision` set to `3` and the desired number of decimal digits, respectively. Note that other parameters, such as feasibility tolerances, are not changed automatically when increasing the precision of computations.

5.2 Technical Improvements

SoPlex has been fully templatized. All old classes, such as `SoPlex`, still remain available. New template classes have been introduced by adding the suffix "base" to the class name, e.g., `SoPlex = SoPlexBase<Real>`. General template functions are now located

[1] Available in source code at `https://developers.google.com/optimization`.

in a `.hpp` file corresponding to the original header. SoPlex now uses the Boost library `program_options` to parse command line arguments, and supports printing all available parameters to the command line with the `--help` option.

Additionally, the SoPlex log prints the total number of violations in the solving log for better understanding of the solving process. The new parameter `int:stattimer` controls time measurement for statistics, and `real:min_markowitz` controls the Markowitz stability threshold.

5.3 New Build Requirements

While Boost is now required to build the SoPlex command line interface, the SoPlex library can still be built without any additional dependencies. In such cases, the multi-precision functionality would not be available. More detailed information on the installation is available in the install guide of SoPlex.

6 Parallel Benders' Decomposition with the UG Framework

The *Ubiquity Generator* (UG) is a generic framework for parallelizing branch-and-bound solvers. The SCIP Optimization Suite contains UG parallelizations of SCIP for both shared and distributed memory computing environments, namely FiberSCIP [75] and ParaSCIP [74]. A more detailed recent overview of the UG framework is given by Shinano [73].

The application of Benders' decomposition (BD) separates an original problem into a master problem and a collection of disjoint subproblems. The solution algorithm associated with BD exposes a simple and intuitive parallelization scheme, especially when there are a large number of subproblems: For each candidate master solution the subproblems can be solved in parallel. This particular parallelization scheme is well suited to the traditional BD implementation, where an iteration of the algorithm involves solving the master problem to optimality before the subproblems are solved to generate cuts. However, the branch-and-cut approach—used by the BD framework in SCIP [58]—reveals alternative parallelization schemes for the BD algorithm.

Numerous methods have been proposed for the parallelization of the branch-and-cut algorithm. However, to the best of the authors knowledge only Langer et al. [52] have proposed the use of parallelization for BD when employing the branch-and-cut approach. In the current release, the parallelization of SCIP by the UG framework has been exploited to develop an alternative parallel BD algorithm.

In Section 4.9.1, the extensions made to the API of SCIP to enable the parallelization of the BD algorithm are presented. These extensions were necessary to support the use of UG to parallelize the branch-and-cut BD algorithm. In particular, the UG framework relies on being able to transfer a node in the branch-and-bound tree from one solver to another. In addition to the necessary information that defines a node in the branch-and-cut tree, all cuts generated by the BD constraint handlers need to be collected and transferred with the node information. The function `SCIPstoreBendersCut` is necessary within the BD cuts plugins for storing the generated cuts. Within the UG framework the functions `SCIPbendersGetStoredCutData` and `SCIPapplyBendersStoredCuts` are called to collect the stored cuts from the current node-processing solver and then applying these cuts to the node-receiving solver, respectively.

To ensure that all necessary plugins are available in each of the solvers, the UG framework uses the copying mechanisms available in SCIP. In a sequential execution of SCIP, the copying of the BD subproblems is not necessary since the second stage constraints are typically relevant for any first stage problem [57]. In the context of parallel branch-and-bound, this is not longer true since potentially many solutions need

to be checked by solving the BD subproblems at the same time. Thus, when copying the BD plugins, i.e. during the executing of the copy callback in a user-defined BD plugin, the subproblems must also be copied. To indicate whether the copy of the BD plugins must be thread-safe, a parameter `threadsafe` is set to `TRUE` and passed to the copying callback. At present, the `threadsafe` parameter is only set to `TRUE` within the UG framework.

To support the use of BD within UG a number of settings files have been provided. The settings `benders-default` and `benders-det` can be used for the default and deterministic execution of UG, respectively. Also, for the default settings the directory `benders_racing` contains settings for performing racing ramp-up with BD. For more details regarding racing ramp-up, the reader is referred to Shinano et al. [74]. Finally, `scip-benders` contains SCIP settings that are useful when using BD with UG.

7 SCIP-SDP

SCIP-SDP is a framework to solve *mixed-integer semidefinite programs* (MISDP) of the following form

$$\begin{aligned} \sup \quad & b^\top y \\ \text{s.t.} \quad & C - \sum_{i=1}^{m} A^i y_i \succeq 0, \\ & \ell_i \leq y_i \leq u_i && \text{for all } i \in [m], \\ & y_i \in \mathbb{Z} && \text{for all } i \in \mathcal{I}, \end{aligned} \tag{23}$$

where $C \in \mathbb{R}^{n \times n}$ and $A^i \in \mathbb{R}^{n \times n}$ are symmetric matrices for all $i \in [m]$. Furthermore, $b \in \mathbb{R}^m$, $\ell \in (\mathbb{R} \cup \{-\infty\})^m$, $u \in (\mathbb{R} \cup \{\infty\})^m$, and $\mathcal{I} \subseteq [m]$ is an index set of integer variables. Here $M \succeq 0$ denotes the fact that the symmetric matrix M is positive semidefinite. Problem (23) is a semidefinite program (SDP) in dual form with additional variable bounds and integrality constraints. A detailed overview of MISDPs is given by Gally [26] and Gally et al. [27]. SCIP-SDP is able to read MISDP instances using an extended SDPA-format or the Conic Benchmark Format (CBF), see `http://cblib.zib.de`.

Version 3.2.0 of SCIP-SDP contains the following new features: Enhanced reading functionality of the CBF format, rank-1 constraints on the matrices, upgrading of quadratic constraints, and several bug fixes.

CBF Format The specification of the CBF-format supports SDPs in dual form (23), but without variable bounds, as well as SDPs in primal form

$$\begin{aligned} \inf \quad & C \bullet X \\ \text{s.t.} \quad & A^i \bullet X = b_i, \quad \text{for all } i \in [m], \\ & X \succeq 0. \end{aligned} \tag{24}$$

Here $A \bullet B = \sum_{i,j=1}^{n} A_{ij} B_{ij}$ for A, $B \in \mathbb{R}^{n \times n}$. Moreover, the CBF-format allows for SDPs with both primal and dual variables, both primal and dual constraints, and a mixed objective function of the form

$$\inf / \sup \{C \bullet X + b^\top y\}.$$

With the new version of SCIP-SDP, reading of SDPs in primal form (24) and mixed form is supported. Since SCIP-SDP internally works with the dual form (23) of an SDP, the primal variable X is replaced by a set of scalar variables X_{ij} with $1 \leq j \leq i \leq n$. The constraints $A^i \bullet X = b_i$, $i \in [m]$, are transformed to linear constraints in the new

Table 5: Specification of rank-1 constraints in an extended CBF file.

`PSDVARRANK1`		`PSDCONRANK1`	
r	number of rank-1 dual constraints	r	number of rank-1 primal variables
m_1		v_1	
$\vdots$	indices of rank-1 dual constraints	$\vdots$	indices of rank-1 primal variables
m_r		v_r	

variables X_{ij}, which are then represented by diagonal entries in the SDP-constraint. Moreover, the constraint $X \succeq 0$ is transformed into a dual constraint

$$\sum_{1 \leq j \leq i \leq n} E^{ij} X_{ij} \succeq 0$$

in the new variables X_{ij}, where E^{ij} is an $n \times n$ matrix with a 1 at entry (i, j) and 0 otherwise. Furthermore, the CBF-reader of SCIP can now parse arbitrary orders of the conic constraints (signs of linear or SDP constraints).

Rank-1 Cosntraints As a second new feature, SCIP-SDP is able to handle rank-1 constraints on both a matrix variable $X \succeq 0$ and a dual constraint $C - \sum_{i=1}^{m} A^i y_i \succeq 0$. This is done by using the fact that a nonzero symmetric positive semidefinite $n \times n$ matrix Z has rank 1 if and only if all of its 2×2 principal minors are zero, see [17]. If a constraint $\operatorname{rank}(X) = 1$ is present in a mixed primal/dual SDP, the quadratic inequalities are

$$X_{ii} X_{jj} - X_{ij}^2 = 0, \tag{25}$$

for all $1 \leq j < i \leq n$. In case a constraint $\operatorname{rank}(C - \sum_{i=1}^{m} A^i y_i) = 1$ is present, the following quadratic equalities are added to the problem during `ConsInitSol`.

$$C_{ii} C_{jj} - C_{ij}^2 - \sum_{k=1}^{m} y_k \big(C_{ii} A_{jj}^k + C_{jj} A_{ii}^k - 2\, C_{ij} A_{ij}^k \big) + \sum_{k,\ell=1}^{m} y_k y_\ell \big(A_{ii}^k A_{jj}^\ell - A_{ij}^k A_{ij}^\ell \big) = 0,$$

for all $1 \leq j < i \leq n$. In order to specify rank-1 constraints for a dual constraint or a primal matrix variable, the CBF-reader of SCIP-SDP supports the two additional keywords `PSDVARRANK1` and `PSDCONRANK1` with the specification given in Table 5.

Upgrading of Quadratic Constraints It is also possible to automatically upgrade quadratic constraints

$$\alpha \leq \sum_{i,j=1}^{n} a_{ij} x_i x_j + \sum_{i=1}^{n} b_i x_i \leq \beta$$

to rank-1 SDP constraints. To this end, variables X_{ij} are added, which should be equal to $x_i \cdot x_j$ or equivalently $xx^\top = X$. This is enforced by the SDP rank-1 constraint

$$\begin{pmatrix} 1 & x^\top \\ x & X \end{pmatrix} \succeq 0, \quad \operatorname{rank} \begin{pmatrix} 1 & x^\top \\ x & X \end{pmatrix} = 1.$$

In fact, if a symmetric matrix has rank 1, there exists a vector $\binom{y_0}{y}$ such that

$$\begin{pmatrix} y_0 \\ y \end{pmatrix} (y_0, y^\top) = \begin{pmatrix} y_0^2 & y_0 y^\top \\ y_0 y & y y^\top \end{pmatrix} = \begin{pmatrix} 1 & x^\top \\ x & X \end{pmatrix}.$$

This implies that $y_0 = \pm 1$, $y = \pm x$. Then $yy^\top = xx^\top = X$, which is the constraint that we want to enforce. Note that the SDP constraint is redudant, but strengthens the relaxation. The quadratic constraint then becomes the linear constraint

$$\alpha \leq \sum_{i,j=1}^{n} a_{ij} X_{ij} + \sum_{i=1}^{n} b_i x_i \leq \beta.$$

This upgrading is enabled with the parameter `constraints/SDP/upgradquadconss`. It is only applied if the number of variables appearing in quadratic constraints is not too big (as specified by the parameter `maxnvarsquadupgd`, default: 1000).

8 Final Remarks

The SCIP Optimization Suite 7.0 release provides significant improvements both in terms of performance and functionality. Thereby, the focus of the release is mainly on MIP solving and extensions.

With the first release of PaPILO included into the SCIP Optimization Suite 7.0, a parallelized, highly effective presolving library is available, which already extends the SCIP presolving routines and will replace SoPlex presolving in a future release. Additionally, both PaPILO 1.0 and SoPlex 5.0 provide new multiprecision support for improved numerical accuracy.

Within SCIP, new features like the decomposition structure and the tree size estimation open new options for future developments. In SCIP 7.0, they already allowed for the inclusion of two primal heuristics based on decomposition structures and dynamic restarts during the tree search. Additionally, SCIP 7.0 comes with many more features improving its performance, including presolving methods, branching rules, and extended conflict analysis and symmetry handling.

Moreover, the performance and usability of the Benders' decomposition framework improved considerably, including two forms of parallelization. Finally, SCIP-SDP provides support for rank-1 SDP constraints which allows for upgrades from quadratic constraints.

These developments combine to a significant speedup on MIPs compared to SCIP 6.0. Since there were only very few changes tailored to MINLPs, no big performance change on this type of problems can be observed. However, MINLP will be a major focus of the next release, which will change the way how nonlinear constraints are handled in SCIP.

Acknowledgements

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. In particular, thanks go to Frédéric Didier for his contributions to the GLOP LPi.

Code 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.

PaPILO was implemented by LG. The improvements to presolve presented in Section 4.1 have been implemented by WC, PG, LG, AG, and DW. The decomposition

structure is due to GH, CT, and KH. The new penalty alternating direction method heuristic was implemented by KH and DW; the modification of GINS and the adaptive diving heuristic are due to GH. The improvements to branching in Section 4.4 have been implemented by GG (degeneracy-aware hybrid branching and lookahead branching), DA and PLB (treemodel scoring functions), MG (vanilla full strong branching), and SV (branching point selection). Locally valid conflicts (Section 4.5) were added by JW. DA, PLB, GH, and MP contributed to the tree size estimation and clairvoyant restarts (Sections 4.6 and 4.7). The extension of the symmetry code (Section 4.8) is due to CH and MP. The updates to the Benders' decomposition framework presented in Section 4.9 have been completed primarily by SJM, with the exception of the *improved handling of MINLPs* which involved a significant contribution from SV. The relaxation-only variables, extended constraint upgrades, and tightening of the LP feasibility tolerance for enforcement have been added by SV. The changes in SoPlex (Section 5) are due to LE. The parallelization of Benders' decomposition using the UG framework, described in Section 6, was completed by YS. The updates to SCIP-SDP were implemented by FM and MP.

References

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

[2] T. Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universität Berlin, 2007.

[3] T. Achterberg. SCIP: Solving Constraint Integer Programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[4] T. Achterberg and T. Berthold. Hybrid branching. In W. J. van Hoeve and J. N. Hooker, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 6th International Conference, CPAIOR 2009*, volume 5547 of *Lecture Notes in Computer Science*, pages 309–311. Springer, 2009.

[5] T. Achterberg, R. E. Bixby, Z. Gu, E. Rothberg, and D. Weninger. Presolve reductions in mixed integer programming. *INFORMS Journal on Computing*, 2019. doi:10.1287/ijoc.2018.0857.

[6] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[7] D. Anderson, G. Hendel, P. Le Bodic, and J. M. Viernickel. Clairvoyant restarts in branch-and-bound search using online tree-size estimation. In *AAAI-19: Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence*, pages 1427–1434. AAAI Press, 2019.

[8] D. Anderson, P. Le Bodic, and K. Morgan. A note on an abstract model for branching and its application to mixed integer programming. *CoRR*, 1909.01472, 2019. URL `http://arxiv.org/abs/1909.01472`.

[9] E. Balas and E. Zemel. An algorithm for large zero-one knapsack problems. *Operations Research*, 28(5):1130–1154, 1980.

[10] C. Beeri, P. A. Bernstein, and N. Goodman. A model for concurrency in nested transactions systems. *J. ACM*, 36(2):230—269, 1989. doi:10.1145/62044.62046.

[11] P. Bendotti, P. Fouilhoux, and C. Rottner. Orbitopal fixing for the full (sub-)orbitope and application to the unit commitment problem, 2018. `http://www.optimization-online.org/DB_FILE/2017/10/6301.pdf`.

[12] T. Berthold. Measuring the impact of primal heuristics. *Operations Research Letters*, 41 (6):611–614, 2013. doi:10.1016/j.orl.2013.08.007.

[13] T. Berthold, G. Gamrath, and D. Salvagnin. Exploiting dual degeneracy in branching. ZIB-Report 19-17, Zuse Institute Berlin, 2019.

[14] Boost. Boost C++ Libraries. `http://www.boost.org`.

[15] L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[16] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[17] C. Chen, A. Atamtürk, and S. S. Oren. A spatial branch-and-cut method for nonconvex QCQP with bounded complex variables. *Mathematical Programming*, 165(2):549–577, 2017.

[18] W.-K. Chen, P. Gemander, A. Gleixner, L. Gottwald, A. Martin, and D. Weninger. Two-row and two-column mixed-integer presolve using hashing-based pairing methods. Technical report, Optimization Online, 2019. `http://www.optimization-online.org/DB_HTML/2019/09/7357.html`.

[19] COIN-OR. CppAD, a package for differentiation of C++ algorithms. `http://www.coin-or.org/CppAD`.

[20] Computational Optimization Research at Lehigh Laboratory (CORAL). MIP instances. `https://coral.ise.lehigh.edu/data-sets/mixed-integer-instances/`. Visited 12/2017.

[21] G. B. Dantzig. Discrete-variable extremum problems. *Operations Research*, 5(2):266–288, 1957.

[22] J. Farkas. Theorie der einfachen Ungleichungen. *Journal für die reine und angewandte Mathematik*, 124:1–27, 1902. URL `http://eudml.org/doc/149129`.

[23] M. Fischetti and M. Monaci. A branch-and-cut algorithm for mixed-integer bilinear programming. *European Journal of Operational Research*, pages 506–514, 2019. doi:10.1016/j.ejor.2019.09.043.

[24] M. Fischetti, I. Ljubić, and M. Sinnl. Redesigning Benders decomposition for large-scale facility location. *Management Science*, 63(7):2146–2162, 2017. doi:10.1287/mnsc.2016.2461.

[25] 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), 2007. doi:10.1145/1236463.1236468.

[26] T. Gally. *Computational Mixed-Integer Semidefinite Programming*. Dissertation, TU Darmstadt, 2019.

[27] 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.

[28] G. Gamrath. *Enhanced Predictions and Structure Exploitation in Branch-and-Bound*. PhD thesis, Technische Universität Berlin, 2020.

[29] 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.

[30] G. Gamrath and C. Schubert. Measuring the impact of branching rules for mixed-integer programming. In *Operations Research Proceedings 2017*, pages 165–170, 2018. doi:10.1007/978-3-319-89920-6_23.

[31] G. Gamrath, T. Koch, A. Martin, M. Miltenberger, and D. Weninger. Progress in presolving for mixed integer programming. *Mathematical Programming Computation*, 7:367–398, 2015. doi:10.1007/s12532-015-0083-5.

[32] G. Gamrath, T. Koch, S. J. Maher, D. Rehfeldt, and Y. Shinano. SCIP-Jack—a solver for STP and variants with parallelization extensions. *Mathematical Programming Computation*, 9(2):231–296, 2017. doi:10.1007/s12532-016-0114-x.

[33] G. Gamrath, T. Berthold, and D. Salvagnin. An exploratory computational analysis of dual degeneracy in mixed-integer programming. *Submitted to the EURO Journal on Computational Optimization*, 2020.

[34] B. Geißler, A. Morsi, L. Schewe, and M. Schmidt. Penalty Alternating Direction Methods for Mixed-Integer Optimization: A New View on Feasibility Pumps. *SIAM Journal on Optimization*, 27:1611–1636, 2017. doi:10.1137/16M1069687.

[35] A. Gleixner, L. Eifler, T. Gally, G. Gamrath, P. Gemander, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, F. Serrano, Y. Shinano, J. M. Viernickel, S. Vigerske, D. Weninger, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 5.0. Technical report, Optimization Online, December 2017. `http://www.optimization-online.org/DB_HTML/2017/12/6385.html`.

[36] A. Gleixner, M. Bastubbe, L. Eifler, T. Gally, G. Gamrath, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. E. Lübbecke, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, C. Schubert, F. Serrano, Y. Shinano, J. M. Viernickel, M. Walter, F. Wegscheider, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 6.0. ZIB-Report 18-26, Zuse Institute Berlin, 2018.

[37] 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. Optimization online preprint: `http://www.optimization-online.org/DB_HTML/2019/07/7285.html`. Submitted to Mathematical Programming Computation, 2019.

[38] G. Hendel. Adaptive large neighborhood search for mixed integer programming. ZIB-Report 18-60, Zuse Institute Berlin, 2018.

[39] G. Hendel, M. Miltenberger, and J. Witzig. Adaptive algorithmic behavior for solving mixed integer programs using bandit algorithms. In B. Fortz and M. Labbé, editors, *Operations Research Proceedings 2018*, pages 513–519. Springer International Publishing, 2019.

[40] G. Hendel, D. Anderson, P. L. Bodic, and M. E. Pfetsch. Estimating the size of branch-and-bound trees. ZIB-Report 20-02 (under preparation), Zuse Institute Berlin, 2020.

[41] C. Hojny. Packing, partitioning, and covering symresacks. Technical report, Optimization Online, 2017. `http://www.optimization-online.org/DB_HTML/2017/05/5990.html`.

[42] 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.

[43] C. C. Holt. Forecasting seasonals and trends by exponentially weighted moving averages. *International Journal of Forecasting*, 20(1):5–10, 2004. doi:10.1016/j.ijforecast.2003.09.015.

[44] Intel. Threading building blocks (TBB). `https://github.com/intel/tbb`, 2020.

[45] Ipopt. Interior Point OPTimizer. `http://www.coin-or.org/Ipopt/`.

[46] T. Junttila and P. Kaski. bliss: A tool for computing automorphism groups and canonical labelings of graphs. `http://www.tcs.hut.fi/Software/bliss/`, 2012.

[47] V. Kaibel and A. Loos. Finding descriptions of polytopes via extended formulations and liftings. In A. R. Mahjoub, editor, *Progress in Combinatorial Optimization*. Wiley, 2011.

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

[49] T. Khaniyev, S. Elhedhli, and F. S. Erenay. Structure detection in mixed-integer programs. *INFORMS Journal on Computing*, 30(3):570–587, 2018.

[50] T. Koch. *Rapid Mathematical Prototyping*. PhD thesis, Technische Universität Berlin, 2004.

[51] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter. MIPLIB 2010. *Mathematical Programming Computation*, 3 (2):103–163, 2011.

[52] A. Langer, R. Venkataraman, U. Palekar, and L. V. Kale. Parallel branch-and-bound for two-stage stochastic integer optimization. In *20th Annual International Conference on High Performance Computing*, pages 266–275, 2013. doi:10.1109/HiPC.2013.6799130.

[53] K. D. Le and J. T. Day. Rolling horizon method: A new optimization technique for generation expansion studies. *IEEE Transactions on Power Apparatus and Systems*, PAS-101(9):3112–3116, Sep. 1982. doi:10.1109/TPAS.1982.317523.

[54] P. Le Bodic and G. L. Nemhauser. An abstract model for branching and its application to mixed integer programming. *Mathematical Programming*, 166(1):369–405, 2017. doi:10.1007/s10107-016-1101-8.

[55] J. Linderoth and S. Wright. Decomposition algorithms for stochastic programming on a computational grid. *Computational Optimization and Applications*, 24(2-3):207–250, 2003. doi:10.1023/A:1021858008222.

[56] A. Loos. *Describing Orbitopes by Linear Inequalities and Projection Based Tools*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, 2010.

[57] S. J. Maher. Enhancing large neighbourhood search heuristics for Benders' decomposition. Technical report, Lancaster University, 2019. http://www.optimization-online.org/DB_HTML/2018/11/6900.html.

[58] S. J. Maher. Implementing the branch-and-cut approach for a general purpose Benders' decomposition framework. Technical report, University of Exeter, 2019. http://www.optimization-online.org/DB_HTML/2019/09/7384.html.

[59] S. J. Maher, G. Desaulniers, and F. Soumis. Recoverable robust single day aircraft maintenance routing problem. *Computers & Operations Research*, 51:130–145, 2014. doi:10.1016/j.cor.2014.03.007.

[60] S. J. Maher, T. Fischer, T. Gally, G. Gamrath, A. Gleixner, R. L. Gottwald, G. Hendel, T. Koch, M. E. Lübbecke, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, S. Schenker, R. Schwarz, F. Serrano, Y. Shinano, D. Weninger, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 4.0. ZIB-Report 17-12, Zuse Institute Berlin, 2017.

[61] F. Margot. Exploiting orbits in symmetric ILP. *Mathematical Programming*, 98(1–3):3–21, 2003. doi:10.1007/s10107-003-0394-6.

[62] F. Margot. Symmetry in integer linear programming. In M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, editors, *50 Years of Integer Programming*, pages 647–686. Springer, 2010.

[63] J. P. Marques-Silva and K. Sakallah. Grasp: A search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, 1999.

[64] MINLPLIB. MINLP library. http://www.minlplib.org.

[65] J. Ostrowski. *Symmetry in Integer Programming*. PhD thesis, Lehigh University, 2008.

[66] J. Ostrowski, J. Linderoth, F. Rossi, and S. Smriglio. Orbital branching. *Mathematical Programming*, 126(1):147–178, 2011. doi:10.1007/s10107-009-0273-x.

[67] O. Y. Özaltın, B. Hunsaker, and A. J. Schaefer. Predicting the solution time of branch-and-bound algorithms for mixed-integer programs. *INFORMS J. on Computing*, 23(3): 392–403, 2011. doi:10.1287/ijoc.1100.0405.

[68] M. E. Pfetsch and T. Rehn. A computational comparison of symmetry handling methods for mixed integer programs. *Mathematical Programming Computation*, 11(1):37–93, 2019. doi:10.1007/s12532-018-0140-y.

[69] A. Ruszczyński. A regularized decomposition method for minimizing a sum of polyhedral functions. *Mathematical Programming*, 35(3):309–333, 1986. doi:10.1007/BF01580883.

[70] T. Santoso, S. Ahmed, M. Goetschalckx, and A. Shapiro. A stochastic programming approach for supply chain network design under uncertainty. *European Journal of Operational Research*, 167(1):96–115, 2005. doi:10.1016/j.ejor.2004.01.046.

[71] L. Schewe, M. Schmidt, and D. Weninger. A Decomposition Heuristic for Mixed-Integer Supply Chain Problems. *Operations Research Letters*, 2020. doi:10.1016/j.orl.2020.02.006.

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

[73] 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 International Publishing, 2018.

[74] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, T. Koch, and M. Winkler. Solving open MIP instances with ParaSCIP on supercomputers using up to 80,000 cores. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 770–779, Los Alamitos, CA, USA, 2016. IEEE Computer Society.

[75] Y. Shinano, S. Heinz, S. Vigerske, and M. Winkler. FiberSCIP – a shared memory parallelization of SCIP. *INFORMS Journal on Computing*, 30(1):11–30, 2018. doi:10.1287/ijoc.2017.0762.

[76] S. Vigerske and A. Gleixner. SCIP: Global optimization of mixed-integer nonlinear programs in a branch-and-cut framework. *Optimization Methods & Software*, 33(3):563–593, 2017. doi:10.1080/10556788.2017.1335312.

[77] J. Witzig, T. Berthold, and S. Heinz. Computational Aspects of Infeasibility Analysis in Mixed Integer Programming. ZIB-Report 19-54, Zuse Institute Berlin, 2019.

[78] J. Witzig, T. Berthold, and S. Heinz. A Status Report on Conflict Analysis in Mixed Integer Nonlinear Programming. In L.-M. Rousseau and K. Stergiou, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 84–94. Springer International Publishing, 2019. doi:10.1007/978-3-030-19212-9_6.

[79] R. Wunderling. *Paralleler und objektorientierter Simplex-Algorithmus*. PhD thesis, Technische Universität Berlin, 1996.

Author Affiliations

Gerald Gamrath
Zuse Institute Berlin, Department of Mathematical Optimization, Takustr. 7, 14195 Berlin and I^{2}DAMO GmbH, Englerallee 19, 14195 Berlin, Germany
E-mail: gamrath@zib.de
ORCID: 0000-0001-6141-5937

Daniel Anderson
Carnegie Mellon University, Computer Science Department, 4902 Forbes Avenue, Pittsburgh PA 15213, USA
E-mail: dlanders@cs.cmu.edu
ORCID: 0000-0002-5853-0472

Ksenia Bestuzheva
Zuse Institute Berlin, Department of Mathematical Optimization, Takustr. 7, 14195 Berlin
E-mail: bestuzheva@zib.de
ORCID: 0000-0002-7018-7099

Wei-Kun Chen
School of Mathematics and Statistics, Beijing Institute of Technology, Beijing 100081, China
E-mail: chenweikun@bit.edu.cn
ORCID: 0000-0003-4147-1346

Leon Eifler
Zuse Institute Berlin, Department of Mathematical Optimization, Takustr. 7, 14195 Berlin
E-mail: eifler@zib.de
ORCID: 0000-0003-0245-9344

Maxime Gasse
Polytechnique Montréal, Department of Mathematics and Industrial Engineering, 2900 Édouard-Montpetit, Montréal (Québec) H3T 1J4, Canada
E-mail: maxime.gasse@polymtl.ca
ORCID: 0000-0001-6982-062X

Patrick Gemander
Friedrich-Alexander Universität Erlangen-Nürnberg, Department Mathematik, Cauerstr. 11, 91058 Erlangen, Germany
E-mail: patrick.gemander@fau.de
ORCID: 0000-0002-0784-6696

Ambros Gleixner
Zuse Institute Berlin, Department of Mathematical Optimization, Takustr. 7, 14195 Berlin
E-mail: gleixner@zib.de
ORCID: 0000-0003-0391-5903

Leona Gottwald
Zuse Institute Berlin, Department of Mathematical Optimization, Takustr. 7, 14195 Berlin

E-mail: gottwald@zib.de
ORCID: 0000-0002-8894-5011

Katrin Halbig
Friedrich-Alexander Universität Erlangen-Nürnberg, Department Mathematik, Cauerstr. 11, 91058 Erlangen, Germany
E-mail: katrin.halbig@fau.de
ORCID: 0000-0002-8730-3447

Gregor Hendel
Zuse Institute Berlin, Department of Mathematical Optimization, Takustr. 7, 14195 Berlin
E-mail: hendel@zib.de
ORCID: 0000-0001-7132-5142

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

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 of Mathematical Optimization, Takustr. 7, 14195 Berlin
E-mail: koch@zib.de
ORCID: 0000-0002-1967-0077

Pierre Le Bodic
Monash University, Faculty of IT, 900 Dandenong Road, Caulfield East VIC 3145, Australia
E-mail: pierre.lebodic@monash.edu
ORCID: 0000-0003-0842-9533

Stephen J. Maher
University of Exeter, College of Engineering, Mathematics and Physical Sciences, Exeter, United Kingdom
E-mail: s.j.maher@exeter.ac.uk
ORCID: 0000-0003-3773-6882

Frederic Matter
Technische Universität Darmstadt, Fachbereich Mathematik, Dolivostr. 15, 64293 Darmstadt, Germany
E-mail: matter@mathematik.tu-darmstadt.de

Matthias Miltenberger
Zuse Institute Berlin, Department of Mathematical Optimization, Takustr. 7, 14195 Berlin
E-mail: miltenberger@zib.de
ORCID: 0000-0002-0784-0964

Erik Mühmer
RWTH Aachen University, Chair of Operations Research, Kackertstr. 7, 52072 Aachen, Germany
E-mail: muehmer@or.rwth-aachen.de

Benjamin Müller
Zuse Institute Berlin, Department of Mathematical Optimization, Takustr. 7, 14195 Berlin
E-mail: benjamin.mueller@zib.de
ORCID: 0000-0002-4463-2873

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

Franziska Schlösser
Zuse Institute Berlin, Department of Mathematical Optimization, Takustr. 7, 14195 Berlin
E-mail: schloesser@zib.de

Felipe Serrano
Zuse Institute Berlin, Department of Mathematical Optimization, Takustr. 7, 14195 Berlin
E-mail: serrano@zib.de
ORCID: 0000-0002-7892-3951

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

Christine Tawfik
Zuse Institute Berlin, Department of Mathematical Optimization, Takustr. 7, 14195 Berlin
E-mail: tawfik@zib.de
ORCID: 0000-0001-5133-0534

Stefan Vigerske
GAMS Software GmbH, c/o Zuse Institute Berlin, Department of Mathematical Optimization, Takustr. 7, 14195 Berlin, Germany
E-mail: svigerske@gams.com

Fabian Wegscheider
Zuse Institute Berlin, Department of Mathematical Optimization, Takustr. 7, 14195 Berlin
E-mail: wegscheider@zib.de

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

Jakob Witzig
Zuse Institute Berlin, Department of Mathematical Optimization, Takustr. 7, 14195 Berlin
E-mail: witzig@zib.de
ORCID: 0000-0003-2698-0767