Enabling Research through the SCIP Optimization Suite 8.0

KSENIA BESTUZHEVA, MATHIEU BESANÇON, WEI-KUN CHEN, ANTONIA CHMIELA, TIM DONKIEWICZ, JASPER VAN DOORNMALEN, LEON EIFLER, OLIVER GAUL, GERALD GAMRATH, AMBROS GLEIXNER, LEONA GOTTWALD, CHRISTOPH GRACZYK, KATRIN HALBIG, ALEXANDER HOEN, CHRISTOPHER HOJNY, ROLF VAN DER HULST, THORSTEN KOCH, MARCO LÜBBECKE, STEPHEN J. MAHER, FREDERIC MATTER, ERIK MÜHMER, BENJAMIN MÜLLER, MARC E. PFETSCH, DANIEL REHFELDT, STEFFAN SCHLEIN, FRANZISKA SCHLÖSSER, FELIPE SERRANO, YUJI SHINANO, BORO SOFRANAC, MARK TURNER, STEFAN VIGERSKE, FABIAN WEGSCHEIDER, PHILIPP WELLNER, DIETER WENINGER, and JAKOB WITZIG

The SCIP Optimization Suite provides a collection of software packages for mathematical optimization centered around the constraint integer programming framework SCIP. The focus of this paper is on the role of the SCIP Optimization Suite in supporting research. SCIP's main design principles are discussed, followed by a presentation of the latest performance improvements and developments in version 8.0, which serve both as examples of SCIP's application as a research tool and as a platform for further developments. Further, the paper gives an overview of interfaces to other programming and modeling languages, new features that expand the possibilities for user interaction with the framework, and the latest developments in several extensions built upon SCIP.

CCS Concepts: • **Theory of computation → Mixed discrete-continuous optimization**; *Parallel algorithms*; **Branch-and-bound**; • **Mathematics of computing → Solvers**; **Mathematical software performance**.

Additional Key Words and Phrases: Constraint integer programming, linear programming, mixed-integer linear programming, mixed-integer nonlinear programming, optimization solver, branch-and-cut, branch-and-price, column generation, parallelization, mixed-integer semidefinite programming

ACM Reference Format:

Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian

Authors' address: Ksenia Bestuzheva, bestuzheva@zib.de; Mathieu Besançon, besancon@zib.de; Wei-Kun Chen, chenweikun@bit.edu.cn; Antonia Chmiela, chmiela@zib.de; Tim Donkiewicz, tim.donkiewicz@rwth-aachen.de; Jasper van Doornmalen, m.j.v.doornmalen@tue.nl; Leon Eifler, eifler@zib.de; Oliver Gaul, oliver.gaul@rwth-aachen.de; Gerald Gamrath, gamrath@zib.de; Ambros Gleixner, gleixner@zib.de; Leona Gottwald, gottwald@zib.de; Christoph Graczyk, graczyk@zib.de; Katrin Halbig, katrin.halbig@fau.de; Alexander Hoen, hoen@zib.de; Christopher Hojny, c.hojny@tue.nl; Rolf van der Hulst, r.p.vanderhulst@utwente.nl; Thorsten Koch, koch@zib.de; Marco Lübbecke, marco.luebbecke@rwth-aachen.de; Stephen J. Maher, s.j.maher@exeter.ac.uk; Frederic Matter, matter@mathematik.tu-darmstadt.de; Erik Mühmer, erik.muehmer@rwth-aachen.de; Benjamin Müller, benjamin.mueller@zib.de; Marc E. Pfetsch, pfetsch@mathematik.tu-darmstadt.de; Daniel Rehfeldt, rehfeldt@zib.de; Steffan Schlein, steffan.schlein@rwth-aachen.de; Franziska Schlösser, schloesser@zib.de; Felipe Serrano, serrano@zib.de; Yuji Shinano, shinano@zib.de; Boro Sofranac, sofranac@zib.de; Mark Turner, turner@zib.de; Stefan Vigerske, svigerske@gams.com; Fabian Wegscheider, wegscheider@zib.de; Philipp Wellner, p.we@fu-berlin.de; Dieter Weninger, dieter.weninger@fau.de; Jakob Witzig, witzig@zib.de.

Manuscript submitted to ACM

Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. 2022. Enabling Research through the SCIP Optimization Suite 8.0. 1, 1 (September 2022), 20 pages. https://doi.org/XXXXXXX.XXXXXXX

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 [34], the presolving library PaPILO, the linear programming solver SoPlex [73], the constraint integer programming solver SCIP [2], which can be used as a fast standalone global solver for mixed-integer linear and nonlinear programs and a flexible branch-cut-and-price framework, the automatic decomposition solver GCG [23], and the UG framework for solver parallelization [57].

All six tools can be downloaded in source code and are freely available for members of noncommercial and academic institutions. Development and bugfix branches of SCIP, SoPlex and PaPILO are mirrored under https://github.com/orgs/scipopt on a daily basis. They are accompanied by several extensions for solving specific problem classes such as the award-winning Steiner tree solver SCIP-Jack [22] and the mixed-integer semidefinite programming (MISDP) solver SCIP-SDP [20]. This paper discusses the capacity of SCIP as a software and research tool and presents the evolving possibilities for working with the SCIP Optimization Suite 8.0, both as a black-box toolbox and as a framework with possibilties of interaction and extension.

Background. SCIP is a branch-cut-and-price framework for solving different types of optimization problems, most importantly, *mixed-integer linear programs* (MILPs) and *mixed-integer nonlinear programs* (MINLPs). MINLPs are optimization problems of the form

$$\begin{aligned} \min \quad & c^\top x \\ \text{s.t.} \quad & Ax \geq b, \\ & \underline{g}_k \leq g_k(x) \leq \overline{g}_k && \text{for all } k \in \mathcal{M}, \\ & \underline{x}_i \leq x_i \leq \overline{x}_i && \text{for all } i \in \mathcal{N}, \\ & x_i \in \mathbb{Z} && \text{for all } i \in \mathcal{I}, \end{aligned} \tag{1}$$

defined by $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m^{(\ell)} \times n}$, $b \in \mathbb{R}^{m^{(\ell)}}$, $\underline{g}, \overline{g} \in \overline{\mathbb{R}}^{m^{(n)}}$, $g : \mathbb{R}^n \to \mathbb{R}^{m^{(n)}}$, $\underline{x}, \overline{x} \in \overline{\mathbb{R}}^n$, the index set of integer variables $\mathcal{I} \subseteq \mathcal{N} := \{1, \ldots, n\}$ and the index set of nonlinear constraints $\mathcal{M} := \{1, \ldots, m^n\}$. We assume that g is specified in algebraic form using basic expressions that are known to SCIP. The usage of $\overline{\mathbb{R}} := \mathbb{R} \cup \{-\infty, \infty\}$ allows for variables that are free or bounded only in one direction (we assume that no variable is fixed to $\pm\infty$). In the absence of nonlinear constraints $\underline{g} \leq g(x) \leq \overline{g}$, the problem becomes an MILP.

SCIP is not restricted to solving MI(N)LPs, 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 MILPs. Later on, this framework allowed for the integration of MINLPs. 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.

The core of SCIP coordinates a central branch-cut-and-price algorithm that is augmented by a collection of plugins. The methods for processing constraints of a given type are implemented in *constraint handler* plugins. The default plugins included in the SCIP Optimization Suite provide tools to solve MI(N)LPs as well as some problems from constraint programming, satisfiability testing and pseudo-Boolean optimization. In this way, advanced methods like primal heuristics, branching rules, and cutting plane separators can be integrated using a pre-defined interface. SCIP comes with many such plugins that enhance MI(N)LP performance, and new plugins can be created by users. This design and solving process is described in more detail by Achterberg [1].

The core solving engine also includes PaPILO, which provides an additional presolving procedure that is called by SCIP, and the linear programming (LP) solver SoPlex which is used by default for solving the LP relaxations within the branch-cut-and-price algorithm. Interfaces to several external LP solvers exist, and new ones can be added by users.

The flexibility of this framework and its design, which is centered around the capacity for extension and customization, are aimed at making SCIP a versatile tool to be used by optimization researchers and practitioners. The possibility to modify the solving process by including own solver components enables users to test their techniques within a general-purpose branch-cut-and-price framework.

The extensions of SCIP that are included in the SCIP Optimization Suite showcase the use of SCIP as a basis for the users' own projects. GCG extends SCIP to automatically detect problem structure and generically apply decomposition algorithms based on the Dantzig-Wolfe or the Benders' decomposition scheme. SCIP-SDP allows to solve mixed-integer semidefinite programs, and SCIP-Jack is a solver for Steiner tree problems. 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.

Examples of Works Using SCIP. A number of works independent of the authors of this paper have employed SCIP as a research tool. Examples of such works include papers on new symmetry handling algorithms [16], branching rules [7] and integration of machine learning with branch-and-bound based MILP solvers [48]. Further application-specific algorithms have been developed based on SCIP, for example, specialized algorithms for solving electric vehicle routing [13] and network path selection [11] problems. Many articles employ SCIP as an MINLP solver for problems such as hyperplanes location [9], airport capacity extension, fleet investment, and optimal aircraft scheduling [15], cryptanalysis problems [17], Wasserstein distance problems [12], and chance-constrained nonlinear programs [32].

Structure of the Paper. The paper is organised as follows. A performance evaluation of SCIP 8.0 and a comparison of its performance to that of SCIP 7.0 is carried out in Section 2. The core solving engine is discussed in Section 3. The interfaces and modeling languages are presented in Section 4. SCIP extensions that are included in the SCIP Optimization Suite are discussed in Section 5, and Section 6 concludes the paper.

For a more detailed description of the new features introduced in SCIP Optimization Suite 8.0, and for the technical details, we refer the reader to the SCIP Optimization Suite 8.0 release report [8].

2 PERFORMANCE OF SCIP 8.0 FOR MILP AND MINLP

In this section, we present computational experiments conducted by running SCIP without parameter tuning or algorithmic variations to assess the performance changes since the 7.0 release. The indicators of interest are the number of solved instances, the shifted geometric mean of the number of branch-and-bound nodes (shift 100 nodes), and the shifted geometric mean of the solving time (shift 1 second).

2.1 Experimental Setup

We use the SCIP Optimization Suite 7.0 as the baseline, including SoPlex 5.0 and PaPILO 1.0, and compare it with the SCIP Optimization Suite 8.0 including SoPlex 6.0 and PaPILO 2.0. Both were compiled using GCC 7.5, use Ipopt 3.12.13

Table 1. Performance comparison for MILP instances

		SCIP 8.0+SoPlex 6.0			SCIP 7.0+SoPlex 5.0			relative	
Subset	instances	solved	time	nodes	solved	time	nodes	time	nodes
all	1708	1478	231.3	3311	1445	271.3	4107	1.17	1.24
affected	1475	1424	173.8	2843	1391	209.7	3611	1.21	1.27
[0,tilim]	1529	1478	154.4	2512	1445	184.6	3167	1.20	1.26
[1,tilim]	1470	1419	185.9	2870	1386	223.8	3647	1.20	1.27
[10,tilim]	1361	1310	248.1	3612	1277	303.1	4661	1.22	1.29
[100,tilim]	1000	949	537.1	7270	916	702.6	10262	1.31	1.41
[1000,tilim]	437	386	1566.2	17973	353	2383.1	31707	1.52	1.76
diff-timeouts	135	84	2072.7	19597	51	5062.1	69354	2.44	3.54
both-solved	1394	1394	119.9	2048	1394	133.8	2330	1.12	1.14

as NLP subsolver built with the MUMPS 4.10.0 numerical linear algebra solver, CppAD 20180000.0 as algorithmic differentiation library, and bliss 0.73 for detecting symmetry. The time limit was set to 7200 seconds in all cases.

The MILP instances are selected from the MIPLIB 2003, 2010, and 2017 [27] as well as the COR@L [37] instance sets and include all instances solved by SCIP 7.0 with at least one of five random seeds or solved by SCIP 8.0 with at least one of five random seeds; this amounts to 347 instances. The MINLP instances are similarly selected from the MINLPLib[1] with newly solvable instances added to the ones solved by SCIP 7.0 for a total of 113 instances.

All performance tests were run on identical machines with Intel Xeon CPUs E5-2690 v4 @ 2.60GHz and 128GB in RAM. A single run was carried out on each machine in a single-threaded mode. Each optimization problem was solved with SCIP using five different seeds for random number generators. This results in a testset of 565 MINLPs and 1735 MILPs. Instances for which the solver reported numerically inconsistent results are excluded from the presented results.

2.2 MILP Performance

Results of the performance runs on MILP instances are presented in Table 1. The "affected" subset contains instances for which the two solver versions show different numbers of dual simplex iterations. Instances in the subsets [t, `tilim`] were solved by at least one solver version within the time limit and took least t seconds to solve with at least one version. "both-solved" and "diff-timeouts" are the subsets of instances that can be solved by both versions and by exactly one version, respectively. "relative" shows the ratio of the shifted geometric mean between the two versions.

The changes introduced with SCIP 8.0 improved the performance on MILPs both in terms of number of solved instances and time. The improvement is more limited on 'both-solved' instances that were solved by both solvers, for which the relative improvement is only of 12 %. This indicates that the overall speedup is more due to newly solved instances than to improvement on instances that were already solved by SCIP 7.0.

2.3 MINLP Performance

With the major revision of the handling of nonlinear constraints, the performance of SCIP on MINLPs has changed considerably compared to SCIP 7.0. The results are summarized in Table 2. More instances are solved by SCIP 8.0 than by SCIP 7.0, and SCIP 8.0 solves the instances for each of these subsets with a shorter shifted geometric mean time. On the 386 instances solved by both versions, SCIP 8.0 requires fewer nodes and less time. The number of instances

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

Table 2. Performance comparison for MINLP

		SCIP 8.0+SoPlex 6.0			SCIP 7.0+SoPlex 5.0			relative	
Subset	instances	solved	time	nodes	solved	time	nodes	time	nodes
all	558	454	39.1	2427	435	45.7	1845	1.17	0.76
affected	487	438	23.5	1748	419	28.4	1456	1.21	0.83
[0,tilim]	503	454	21.7	1585	435	25.9	1326	1.19	0.84
[1,tilim]	375	326	56.1	3994	307	71.0	3113	1.27	0.78
[10,tilim]	293	244	121.6	7450	225	169.3	5393	1.39	0.72
[100,tilim]	195	146	307.6	14204	127	433.9	6696	1.41	0.47
[1000,tilim]	153	104	466.9	23425	85	565.3	8382	1.21	0.36
diff-timeouts	117	68	451.4	29142	49	461.8	6275	1.02	0.22
both-solved	386	386	8.2	609	386	10.4	806	1.27	1.32

solved by only one of the two versions (diff-timeouts) is much higher than reported in previous release reports with similar experiments, with 68 instances solved only by SCIP 8.0 and 49 instances solved only by SCIP 7.0. A performance evaluation that focuses only on the changes in handling nonlinear constraints is given in Section 3.1.5.

3 THE CORE SOLVING ENGINE

This section presents the core solving engine, which includes the CIP solver SCIP, the MILP presolving library PaPILO, and the LP solver SoPlex. It discusses SCIP's MINLP framework in Section 3.1, which was completely reworked in the 8.0 release, and demonstrates the possibilities for implementing user's own methodsusing the examples of two areas that saw improvement with the 8.0 release, namely symmetry handling and primal heuristics in Sections 3.2 and 3.3.

The full list of new features introduced in SCIP 8.0 is the following: a new framework for handling nonlinear constraints, symmetry handling on general variables and improved orbitope detection, a new separator for mixing cuts, improvements to decomposition-based heuristics, the option to apply the mixed integer rounding procedure when generating optimality cuts in the Benders' decomposition framework, a new plugin type that enables users to include their own cut selection rules into SCIP, and several technical improvements.

Further, the section provides an overview of the presolving library PaPILO and the LP solver SoPlex in Sections 3.4 and 3.5, and presents the new dual postsolving feature in PaPILO, which allowed for it to be integrated into SoPlex.

3.1 SCIP's New MINLP Framework

A new framework for handling nonlinear constraints was introduced with the SCIP 8.0 release. The main motivation for this change is twofold: First, it aims at increasing the reliability of the solver and alleviating numerical issues that arose from problem reformulations. Second, the new design of the nonlinear framework reduces the ambiguity of expression and structure types by implementing different kinds of plugins for low-level expressions that define expressions, and high-level structures that add functionality for particular, often overlapping structures.

The main components of the new framework are the following: plugins representing expressions; a reimplementation of the constraint handler for nonlinear constraints, `cons_nonlinear`; nonlinear handler plugins that provide functionality for high-level structures; a revision of the primal heuristic that solves NLP subproblems; revised interfaces to NLP solvers; and revised interface to an automatic differentiation library. Moreover, SCIP 8.0 contains cutting plane separators that work on nonlinear structures and interact with `cons_nonlinear`.

3.1.1 New Expressions Framework. Algebraic expressions are well-formed combinations of constants, variables, and various algebraic operations such as addition, multiplication, exponentiation, that are used to describe mathematical functions. In SCIP, they are represented by a directed acyclic graph with nodes representing variables, constants, and operators and arcs indicating the flow of computation.

With SCIP 8.0, the expression system has been completely rewritten. Proper SCIP plugins, referred to as *expression handlers*, are now used to define all semantics of an operator. These expression handlers support more callbacks than what was available for user-defined operators before. Furthermore, much ambiguity and complexity is avoided by adding expression handlers for basic operations only. High-level structures such as quadratic functions can still be recognized, but are no longer made explicit by a change in the expression type.

3.1.2 New Handler for Nonlinear Constraints. For SCIP 8.0, the constraint handler for nonlinear constraints, `cons_nonlinear`, has been rewritten and constraint handlers for quadratic, second-order cone, absolute power, and bivariate constraints have been removed. Some functionalities of the removed constraint handlers have been reimplemented in other plugins.

An initial motivation for rewriting `cons_nonlinear` was a numerical issue which was caused by explicit constraint reformulation in earlier versions. Such a reformulation can lead to a difference in constraint violation estimation in the original and reformulated problems and, in particular, to a solution being feasible for the reformulated problem and infeasible for the original problem. For example, this occurs in a problem where the constraint $\exp(\ln(1000)+1+x\,y) \leq z$ is reformulated as $\exp(w) \leq z$, $\ln(1000) + 1 + x\,y = w$. On the MINLPLib library, this issue occurred for 7% of instances.

The purpose of the reformulation is to enable constructing a linear relaxation. In this process, nonlinear functions are approximated by linear under- and overestimators. Since the formulas that are used to compute these estimators are only available for "simple" functions, new variables and constraints were introduced to split more complex expressions into adequate form [64, 70].

A trivial attempt to solve the issue of solutions not being feasible in the original problem would have been to add a feasibility check before accepting a solution. However, if a solution is not feasible, actions to resolve the violation of original constraints need to be taken, such as a separating hyperplane, a domain reduction, or a branching operation. Since the connection from the original to the presolved problem was not preserved, it would not have been clear which operations on the presolved problem would help best to remedy the violation in the original problem.

Thus, the new constraint handler aims to preserve the original constraints by applying only transformations that, in most situations, do not relax the feasible space when taking tolerances into account. The reformulations that were necessary for the construction of a linear relaxation are not applied explicitly anymore, but handled implicitly by annotating the expressions that define the nonlinear constraints. Another advantage of this approach is a clear distinction between the variables that were present in the original problem and the variables added for the reformulation. With this information, branching is avoided on variables of the latter type. Finally, it is now possible to exploit overlapping structures in an expression simultaneously.

3.1.3 Extended Formulations. Consider problems of the form (1), where the set of nonlinear constraints is non-empty, and some constraints may be nonconvex. SCIP solves such problems to global optimality via a spatial branch-and-bound algorithm. Important parts of the algorithm are presolving, domain propagation, linear relaxation, and branching. For domain propagation and linear relaxation, extended formulations are used which are obtained by introducing *slack variables* and replacing sub-trees of the expressions that define nonlinear constraints by *auxiliary variables*.

These extended formulations have the following form:

$$\begin{aligned} \min\quad & c^\top x, \\ \text{s.t.}\quad & h_i(x, w_{i+1}, \dots, w_m) = w_i, \qquad\qquad i = 1, \dots, m, \\ & \underline{x} \le x \le \overline{x},\ \underline{w} \le w \le \overline{w},\ x_{\mathcal{I}} \in \mathbb{Z}^{\mathcal{I}}. \end{aligned} \tag{MINLP$_{\text{ext}}$}$$

Here, $w_1, \dots, w_m$ are slack variables, and $h_i := g_i$ for $i = 1, \dots, m$. For each function h_i, subexpressions f may be replaced by new auxiliary variables $w_{i'}$, $i' > m$, and new constraints $h_{i'}(x) = w_{i'}$ with $h_{i'} := f$ are added. For the latter, subexpressions may be replaced again. The result is referred to by $h_i(x, w_{i+1}, \dots, w_m)$ for any $i = 1, \dots, m$. That is, to simplify notation, w_{i+1} is used instead of $w_{\max(i,m)+1}$.

Example of an Extended Formulation. Consider constraint $\log(x)^2 + 2\log(x)\,y + y^2 \le 4$. SCIP may replace $\log(x)$ by an auxiliary variable w_2, since this results in a quadratic form $w_2^2 + 2w_2y + y^2$, which is both bivariate and convex, the former being well suited for domain propagation and the latter being beneficial for linearization. Therefore, the following extended formulation may be constructed:

$$\begin{aligned} h_1(x, y, w_2) &:= (w_2)^2 + 2w_2y + y^2 = w_1, \\ h_2(x, y) &:= \log(x) = w_2,\ w_1 \le 4. \end{aligned}$$

3.1.4 Structure Handling. The construction of extended formulations is based on the information on what algorithms are available for analyzing expressions of a specific structure. Following the spirit of the plugin-oriented design of SCIP, these algorithms are added as separate plugins, referred to as *nonlinear handlers*. Nonlinear handlers can detect structures in expressions and provide domain propagation and linear relaxation algorithms that act on these structures. Unlike other plugins in SCIP, nonlinear handlers are managed by `cons_nonlinear` and not the SCIP core.

Nonlinear handlers for the following expression types are available in SCIP: quadratic expressions defined as sums where at least one term is either a product of two expressions or a square expression, bilinear expressions, convex and concave expressions, quotient expressions of the form $(ay_1 + b)/(cy_2 + d) + e$, and expressions defined in terms of semi-continuous variables. The second-order cone (SOC) nonlinear handler provides separation for SOC constraints. Finally, the default nonlinear handler ensures that there always exist domain propagation and linear under/overestimation routines for an expression and employs callbacks of expression handlers to provide the necessary functionalities.

Additional structures can be recognized for generating cutting planes to strengthen LP relaxations. Such structures are handled by separator plugins. While separators are not restricted to nonlinear structures, the following separators were introduced in SCIP 8.0 that work on MINLPs: the Reformulation-Linearization technique (RLT) [4–6] separator adds RLT cuts for bilinear products and can additionally reveal linearized products between binary and continuous variables; the principal minor separator works on a matrix $X = xx^\top$, where entries X_{ij} represent auxiliary variables corresponding to x_ix_j, and enforces that principle 2×2 minors are PSD; and the intersection cuts separator for rank-1 constraints (disabled by default) adds cuts derived from the condition that any 2×2 minor of X has determinant 0.

3.1.5 Performance Impact of Updates for Nonlinear Constraints. While Section 2.3 compared the performance of SCIP 7.0 and SCIP 8.0, this section takes a closer look at the effect of replacing only the handling of nonlinear constraints in SCIP. That is, here the following two versions of SCIP are compared:

classic: the main development branch of SCIP as of 23.08.2021; nonlinear constraints handled as in SCIP 7.0;

Table 3. Comparison of performance of SCIP with classic versus new handling of nonlinear constraints on MINLPLib.

Subset	instances	metric	classic	new	both
all	5034	solution infeasible	481	49	20
		failed	143	70	18
		solved	2929	3131	2742
		time limit	1962	1833	1598
		memory limit	0	0	0
clean	4839	fastest	3733	3637	2531
		mean time	75.9s	70.3s	
		mean nodes	2543	2601	

new: as classic, but with the handling of nonlinear constraints replaced as detailed in this section and symmetry detection extended to handle nonlinear constraints (see Section 3.2).

SCIP has been build with GCC 7.5.0 and uses PaPILO 1.0.2, bliss 0.73, CPLEX 20.1.0.1 as LP solver, Ipopt 3.14.4, CppAD 20180000.0 and Intel MKL 2020.4.304 for linear algebra (LAPACK). Ipopt uses the same LAPACK and HSL MA27 as linear solver. All runs are carried out on machines with Intel Xeon CPUs E5-2660 v3 @ 2.60GHz and 128GB RAM in a single-threaded mode. A time limit of one hour, a memory limit of 100000MB, an absolute gap tolerance of 10^{-6}, and a relative gap tolerance of 10^{-4} are set. All 1678 instances of MINLPLib (version 66559cbc from 2021-03-11) that can be handled by both versions are used. Note that MINLPLib is not designed to be a benchmark set, since, for example, some models are overrepresented. For each instance, two additional runs were conducted where the order of variables and constraints were permuted. Thus, in total 5034 jobs were run for each version.

Table 3 summarizes the results. A run is considered as failed if the reported primal or dual bound conflicts with best known bounds for the instance, the solver aborted prematurely due to a fatal error, or the solver did not terminate at the time limit. Runs where the final solution is not feasible are counted separately. With the new version, for much fewer instances the final incumbent is not feasible for the original problem, that is, the issue discussed in Section 3.1.2 has been resolved. For the remaining 49 instances, typically small violations of linear constraints or variable bounds occur. Furthermore, the reduction in "failed" instances by half shows that the new version is more robust regarding the computation of primal and dual bounds. Finally, the new version solves about 400 additional instances in comparision to the classic one, but also no longer solves about 200 instances within the time limit.

Subset "clean" refers to all instances where both versions did not fail, i.e., either solved to optimality or stopped due to the time limit. We count a version to be "fastest" on an instance if it is not more than 25% slower than the other version. Mean times were computed as explained in the beginning of Section 2. Due to the increase in the number of solved instances, a reduction in the mean time with the new version on subset "clean" can be observed, even though the new version is fastest on less instances than the classic one.

Figure 1 shows performance profiles that compare both versions w.r.t. the time to solve an instance and the gap at termination. The time comparison visualizes what has been observed in Table 3: the new version solves more instances, but can be slower. The gap comparison shows that on instances that are not solved, often the new version produces a smaller optimality gap than the classic version.

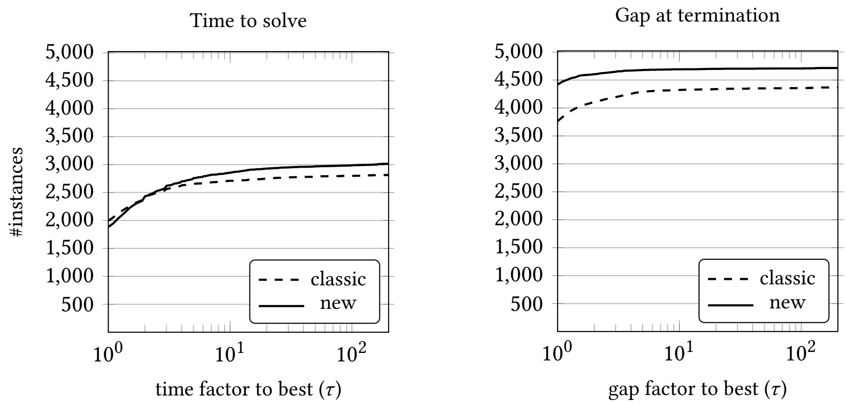

Fig. 1. Performance profiles with classic versus new handling of nonlinear constraints, showing the number of instances for which the corresponding version was at most τ times worse (regarding time (left) or gap at termination (right)) than the best of both versions. For the time plot, instances that were solved to optimality are considered. For the gap plot, instances that did not fail are considered.

3.2 Improvements in Symmetry Handling

Symmetries are known to have an adverse effect on the performance of MI(N)LP solvers due to symmetric subproblems being treated repeatedly without providing new information to the solver. Since detecting all symmetries is $\mathcal{NP}$-hard [41], SCIP only detects symmetries that keep the formulation invariant.

SCIP's symmetry handling framework can be used both as a black box and research tool. In the black box approach, SCIP automatically detects and handles symmetries. If symmetries are known, users can tell SCIP about them by adding specialized constraints. Customized code can include such constraints via API functions, but also black box SCIP can be informed about symmetries via parsing them from files in SCIP's CIP format. Moreover, SCIP facilitates research on symmetries as it stores all symmetry information centrally in the `symmetry` propagator and provides implementations of basic symmetry operations such as stabilizer computations.

For a permutation γ of the variable index set $\{1, \ldots, n\}$ and a vector $x \in \mathbb{R}^n$, we define $\gamma(x) = (x_{\gamma^{-1}(1)}, \ldots, x_{\gamma^{-1}(n)})$. We say that γ is a *symmetry* of (1) if the following holds: $x \in \mathbb{R}^n$ is feasible for (1) if and only if $\gamma(x)$ is feasible, and $c^\top x = c^\top \gamma(x)$. The set of all symmetries forms a group Γ, the *symmetry group* of (1). If Γ is a product group $\Gamma = \Gamma_1 \otimes \cdots \otimes \Gamma_k$, the variables affected by one factor of Γ are not affected by any other factor. In this case, SCIP can apply different symmetry handling methods for each factor. The sets of all variables affected by a single factor are called components.

SCIP 7.0 was only able to handle symmetries of binary variables in MILPs using two paradigms: a constraint-based approach or the pure propagation-based approach *orbital fixing* [39, 40, 46]. For a symmetry γ, the constraint-based approach enforces that the variable vector x is lexicographically not smaller than $\gamma(x)$. This is implemented via three different constraint handler plugins. For single permutations γ, the *symresack* and *orbisack* constraint handlers use separation and propagation [28] techniques for enforcing the lexicographic requirement, also c.f. [31]. Additionally, if an entire factor Γ_i of Γ has a special structure, the *orbitope* constraint handler applies specialized techniques [21].

SCIP 8.0 extends the symmetry handling framework. First, it allows to detect symmetries in MINLPs [72]. Second, in SCIP 8.0, symmetries of general variables can be handled by inequalities derived from the Schreier-Sims table (SST cuts) [36, 54]. These inequalities are based on a list of *leaders* $\ell_1, \ldots, \ell_k$ together with suitably defined *orbits* $O_1, \ldots, O_k$, leading to inequalities $x_{\ell_i} \geq x_j$, $j \in O_i$, $i \in \{1, \ldots, k\}$. Users have a high degree of flexibility to control the selection of orbits and can thus select the most promising symmetry handling strategy. Third, orbitope detection has been extended to also detect suborbitopes, i.e., parts of the symmetry group that allow to apply orbitopes. Since adding suborbitopes did not turn out to always be beneficial, SCIP adds suborbitopes according to a strategy that can combine suborbitopes and SST cuts; adding SST cuts can be controlled by a user via parameters.

Furthermore, SCIP 8.0 contains improvements of previously available methods. First, if orbisack constraints interact with set packing or partitioning constraints in a certain way, they are automatically upgraded to orbitopes. This upgrade has been made more efficient. Second, the running time of the separation routine of cover inequalities for symresacks has been improved from quadratic to linear by using the observation from [28] that minimal cover inequalities for symresacks can be separated by merging connected components of an auxiliary graph. The new implementation exploits that its connected components are either paths or cycles. Finally, propagation routines of the symresack and orbisack constraint handler now find all variable fixings that can be derived from local variable bound information.

3.3 Primal Decomposition Heuristics

Most MILPs have sparse constraint matrices for which 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 or heuristics can be designed exploiting the underlying structure and yielding smaller, easier problems. In this sense, a so-called *decomposition* identifies subsets of rows and columns that are only linked to each other via a set of linking rows and/or linking columns, but are otherwise independent.

A decomposition consisting of $k \in \mathbb{N}$ blocks is a partition $\mathcal{D} := (D^{\text{row}}, D^{\text{col}})$ with $D^{\text{row}} := (D^{\text{row}}_1, \dots, D^{\text{row}}_k, L^{\text{row}})$, $D^{\text{col}} := (D^{\text{col}}_1, \dots, D^{\text{col}}_k, L^{\text{col}})$ of the rows/columns of the constraint matrix A into $k+1$ pieces each, whereby it holds for all $i \in D^{\text{row}}_{q_1}$, $j \in D^{\text{col}}_{q_2}$ that $a_{i,j} \neq 0$ implies $q_1 = q_2$. Rows L^{row} and columns L^{col}, which may be empty, are called *linking rows and columns*, respectively.

In general, there is no unique way to decompose an MILP, and different decompositions might lead to different solver behaviors. Users might be aware of decompositions and know which are most useful for a specific problem. Therefore, since version 7.0 it is possible to pass user decompositions to SCIP [21]. A decomposition structure can be created using the SCIP API, assigning labels to variables and/or constraints, and calling automatic label computation procedures if necessary. Alternatively, SCIP also provides a file reader for decompositions in constraints.

In SCIP 7.0, the Benders decomposition framework and the heuristic Graph Induced Neighborhood Search were extended to exploit user-provided decompositions, and a first version of the heuristic Penalty Alternating Direction Method (PADM) [25, 55] was introduced. SCIP 8.0 comes with an improvement of PADM and provides another decomposition heuristic Dynamic Partition Search (DPS) [8].

Improvement of Penalty Alternating Direction Method. PADM splits an MINLP into several subproblems according to a given decomposition $\mathcal{D}$ with linking variables only, whereby the linking variables get copied and the differences are penalized. Then the subproblems are alternatingly solved. For faster convergence, the objective function of each subproblem has been replaced by a penalty term, and this replacement can lead to arbitrarily bad solutions. Therefore, PADM has been extended by the option to improve a found solution by reintroducing the original objective function.

Dynamic Partition Search. The new primal construction heuristic DPS requires a decomposition with linking constraints only. The linking constraints and their sides are split by introducing vectors $p_q \in \mathbb{R}^{L^{\text{row}}}$ for each block $q \in \{1, \dots, k\}$, where $\mathbb{R}^{L^{\text{row}}}$ denotes the space of vectors with components indexed by L^{row}, and requiring that the following holds:

$$\sum_{q=1}^{k} p_q = b_{[L^{\text{row}}]}. \tag{2}$$

To obtain information on subproblem infeasibility and speed up the solving process, the objective function is replaced by a weighted sum of slack variables $z_q \in \mathbb{R}_+^{L^{\text{row}}}$. For penalty parameter $\lambda \in \mathbb{R}_{>0}^{L^{\text{row}}}$, each subproblem q has the form

$$\begin{aligned}
\min \quad & \lambda^\top z_q, \\
\text{s.t.} \quad & A_{[D_q^{\text{row}}, D_q^{\text{col}}]}\, x_{[D_q^{\text{col}}]} \geq b_{[D_q^{\text{row}}]}, \\
& \underline{x}_i \leq x_i \leq \overline{x}_i && \text{for all } i \in \mathcal{N} \cap D_q^{\text{col}}, \\
& x_i \in \mathbb{Z} && \text{for all } i \in \mathcal{I} \cap D_q^{\text{col}}, \\
& A_{[L^{\text{row}}, D_q^{\text{col}}]}\, x_{[D_q^{\text{col}}]} + z_q \geq p_q, \\
& z_q \in \mathbb{R}_+^{L^{\text{row}}}.
\end{aligned} \tag{3}$$

From (3), it is apparent that the correct choice of p_q plays a central role. For this reason, we refer to $(p_q)_{q \in \{1,\dots,k\}}$ as a *partition* of $b_{[L^{\text{row}}]}$. The method starts with an initial partition fulfilling (2). Then it is checked whether this partition will lead to a feasible solution by solving k independent subproblems (3) with fixed p_q. If the current partition does not correspond to a feasible solution, then the partition gets updated, so that (2) still holds. These steps are repeated. Similarly to PADM, it is possible to improve the found solution by reoptimizing with the original objective function.

3.4 PaPILO

The C++ library PaPILO provides presolving routines for (MI)LP problems and was introduced with the SCIP Optimization Suite 7.0 [21]. PaPILO can be integrated into MILP solvers or used as a standalone presolver. As a standalone presolver it provides presolving and postsolving routines. Hence, it can be used to a) provide presolving for new solving methods and b) generate presolved instances so that different solvers can be benchmarked independently of their own presolvers. Thus, the performance/behavior of the actual solver can be evaluated and compared more precisely.

PaPILO's transaction-based design allows presolvers to run in parallel without requiring expensive copies of the problem and without special synchronizations. Instead of applying results immediately, presolvers return their reductions to the core, where they are applied in a deterministic, sequential order. Validity of every reduction to the modified problem is checked to avoid applying conflicting reductions.

Presolving deletes variables from the original problem by fixing, substituting, and aggregating variables. After solving the reduced problem, its solution does not contain any information on missing variables. To restore the solution values of these variables and obtain a feasible solution of the original problem, corresponding data needs to be stored during the presolving process. The process of recalculating the original solution from the reduced one is called postsolving or post-processing [3]. Until version 1.0.2, PaPILO supported only postsolving primal solutions for LPs. In the latest version, PaPILO supports postsolving also for dual solutions, reduced costs, slack variables of the constraints, and the basic status of the variables and constraints for the majority of the LP presolvers.

3.5 SoPlex

SoPlex is a simplex-based LP solver and an essential part of the optimization suite, since is the default LP-solver for SCIP. In addition to all the essential features of a state-of-the-art LP solver such as scaling, exploitation of sparsity, or presolving, SoPlex also supports an option for 80bit extended precision and an iterative refinement algorithm to produce high-precision solutions. This enables SoPlex to also compute exact rational solutions to LPs, using either continued fraction approximations or a symbolic LU factorization.

The support of postsolving of dual LP solutions and basis information in PaPILO makes it possible to integrate PaPILO fully as a presolving library into SoPlex. In version 6.0 of SoPlex, PaPILO is available as an additional option for presolving. The previous presolving implementation continues to be the default.

4 MODELING LANGUAGES AND INTERFACES

There are many interfaces to SCIP from different programming and modeling languages. These interfaces allow users to programmatically call SCIP with an API close to the C one or leverage a higher-level syntax.

The AMPL interface has been rewritten and moved to the main SCIP library and executable. With the SCIP Optimization Suite 8.0, there exists a C wrapper for SoPlex, updated GAMS interfaces for SoPlex and SCIP, a Julia package `SCIP.jl`, a basic Java interface JSCIPOpt, a new Matlab interface for SCIP 8.0 and SCIP-SDP based on the `OPTI Toolbox` by Jonathan Currie, and the Python interface PySCIPOpt which can now also be installed as a Conda package.

The modeling language Zimpl allows for MI(N)LPs to be written and translated into some file formats supported by SCIP. Zimpl 3.5.0 allows quadratic objective functions in addition to previously supported linear objective functions, and can write suitable instances as Quadratic Unconstrained Binary Optimization problems.

5 EXTENSIONS

5.1 The GCG Decomposition Solver

SCIP allows implementing tailored decomposition-based algorithms. Complementary to this, GCG turns SCIP into a generic decomposition-based *solver* for MILPs. While GCG's focus is on Dantzig-Wolfe reformulation (DWR) and Lagrangian decomposition, Benders decomposition (BD) is also supported. The philosophy behind GCG is that decomposition-based algorithms can be routinely applied to MILPs without the user's interaction or even knowledge. To this end, GCG automatically detects a model structure that admits a decomposition and performs the corresponding reformulation. This results in a master problem and one or several subproblems, which are usually formulated as MILPs. Based on the reformulation, the linear relaxation in every node is solved by column generation (in the DWR case) and Benders cut generation (in the BD case). GCG features primal heuristics and separation of cutting planes, several of which are adapted from SCIP, but some are tailored to the decomposition situation in which both an original and a reformulated model are available. As a research tool, GCG can be used to quickly assess the potential of a decomposition-based algorithm for any problem for which a compact MILP formulation is available. This allows evaluating the performance of an algorithmic idea with a single generic implementation, but across many different applications. In what follows, we describe some enhancements in the GCG 3.5 release.

5.1.1 Detection Loop Refactoring. Decomposition-based algorithms rely on model structures, cf. Section 3.3. For automatic identification of such structures, GCG features a modular detection loop. *Detectors* iteratively assign roles like "master" or "block" to variables and/or constraints. This way, usually many different potential decompositions are found. We refer to the SCIP Optimization Suite 6.0 release report [26] for a more detailed overview. Detectors are implemented as plugins such that new ones can be added conveniently. In every round, each detector works on existing (but possibly empty) partial or complete decompositions. An empirically very successful detection concept builds on the classification of constraints and variables, which is performed prior to the actual detection process, using *classifiers*.

5.1.2 Branching. In GCG, two general branching rules are implemented (branching on original variables [71] and Vanderbeck's generic branching [69]) as well as one rule that applies only to set partitioning master problems (Ryan and Foster branching [53]). While these rules differ significantly, the general procedure has two common stages: First, one determines the set of candidates we could possibly branch on (called the branching rule). Second, the *branching candidate selection heuristic* selects one of the candidates. GCG previously contained pseudo cost, most fractional, and random branching as selection heuristics for original variable branching, and first-index branching for Ryan-Foster and Vanderbeck's generic branching. In GCG 3.5, new strong branching-based selection heuristics are added [24].

5.1.3 Python Interface. With GCG 3.5 we introduce PyGCGOpt which extends SCIP's existing Python interface [38] for GCG and is distributed independently from the optimization suite[2]. All existing functionality for MILP modeling is inherited from PySCIPOpt; therefore, any MILP modeled in Python can be solved with GCG without additional effort. The interface supports specifying custom decompositions and exploring automatically detected decompositions, and plugins for detectors and pricing solvers can be implemented in Python.

5.1.4 Visualization Suite. Visualizations of algorithmic behavior can yield understanding and intuition for interesting parts of a solving process. With GCG 3.5, we include a Python-based *visualization suite* that offers visualization scripts to show processes and results related to detection, branching, or pricing, among others. We highlight two features:

(1) Reporting functionality: A *decomposition report* offers an overview of all decompositions that GCG found for a single run. For different runs, GCG 3.5 offers two reports: A *testset report* shows data and graphics for each single run of one selected test set. A *comparison report* allows to compare two or more runs on the same test set.
(2) Jupyter notebook: data produced for the reports can be read, cleaned, filtered, and visualized interactively.

5.2 SCIP-SDP

SCIP-SDP is an MISDP solver and a platform for implementing methods for solving MISDPs. It was initiated by Sonja Mars and Lars Schewe [42], and then continued by Gally et al. [20] and Gally [19]. New results and methods mainly concerning presolving and propagation are presented in [43]. SCIP-SDP features interfaces to SDP-solvers DSDP, Mosek, and SDPA.

SCIP-SDP implements an SDP-based branch-and-bound method, which solves a continuous SDP relaxation in each node. It incorporates plugins such as primal heuristics, presolving and propagation methods, and file readers. There is also an option to solve LP relaxations in each node of the branch-and-bound tree and generate eigenvector cuts, see Sherali and Fraticelli [56]. This is sometimes faster than solving SDPs in every node. These two options can also be run concurrently if the parallel interface `TPI` of SCIP is used. There also is a Matlab interface to SCIP-SDP.

Moreover, SCIP-SDP can handle rank-1 constraints, that is, the requirement that the resulting matrix $A(y)$ in (**??**) has rank 1. For such a constraint, quadratic constraints are added, modeling that all 2×2-minors of $A(y)$ are zero [14].

Before we present some computational results, let us add some words of caution. Although SCIP-SDP is numerically quite robust, accurately solving SDPs is more demanding than solving LPs. This can lead to wrong results on some instances[3], and the results often depend on the tolerances. Moreover, the SDP-solvers use relative tolerances, while SCIP-SDP uses absolute tolerances. Finally, for Mosek, we use a slightly tighter feasibility tolerance than in SCIP-SDP.

Table 4 shows a comparison between SCIP-SDP 3.2 and 4.0 on the same testset as used by Gally et al. [20], which consists of 194 instances; the changes between SCIP-SDP 4.0 and 3.2 are presented in more detail in [8, 43]. Reported are

[2] https://github.com/scipopt/PyGCGOpt
[3] For instance, in seldom cases, the dual bound might exceed the value of a primal feasible solution.

Table 4. Performance comparison of SCIP-SDP 4.0 vs. SCIP-SDP 3.2

	# opt	# nodes	time [s]
SCIP-SDP 3.2	185	617.3	42.9
SCIP-SDP 4.0	187	497.3	26.6

the number of optimally solved instances, as well as the shifted geometric means of the number of processed nodes and the CPU time in seconds. We use Mosek 9.2.40 for solving the continuous SDP relaxations. The tests were performed on a Linux cluster with 3.5 GHz Intel Xeon E5-1620 Quad-Core CPUs, having 32 GB main memory and 10 MB cache. All computations were run single-threaded and with a timelimit of one hour.

As can be seen from the results, SCIP-SDP 4.0 is considerably faster than SCIP-SDP 3.2, but we recall that we have relaxed the tolerances. Nevertheless, we conclude that SCIP-SDP has significantly improved since the last version.

5.3 SCIP-Jack: Solving Steiner Tree and Related Problems

Given an undirected, connected graph, edge costs and a set of *terminal* vertices, the *Steiner tree problem in graphs* (SPG) asks for a tree of minimum weight that covers all terminals. The SPG is a fundamental $\mathcal{NP}$-hard problem [33] and one of the most studied problems in combinatorial optimization.

SCIP-Jack, an exact SPG-solver, is built on the branch-and-cut framework provided by SCIP and makes extensive use of its plugin-based design. At the heart of the implementation is a constraint handler that separates violated constraints, most importantly the so-called *directed Steiner cuts*, which are separated by a specialized maximum-flow algorithm [49]. The implementation includes a variety of additional SCIP plugins, such as heuristics, propagators, branching rules and relaxators. Finally, the use of SCIP provides significant flexibility in the model to be solved, for example it is easily possible to add additional constraints. In this way, SCIP-Jack can solve not only the SPG, but also 14 related problems.

The SCIP Optimization Suite 8.0 contains the new SCIP-Jack 2.0[4], which comes with major improvements on most problem classes it can handle and outperforms the SPG solver by Polzin and Vahdati [47, 68], which had remained unchallenged for almost 20 years, on almost all nontrivial benchmark testsets from the literature [51].

Figure 2 provides computational results on the instances from Tracks A and B of the PACE Challenge 2018 [10]. We use Gurobi 9.5 (*Commercial*), the best other solver from the PACE Challenge (*SPDP* [29]), and SCIP-Jack with SoPlex (*SCIPJ/spx*) and Gurobi 9.5 (*SCIPJ/grb*) as LP solvers. A timelimit of one hour was set. Average times are given as arithmetic means with time-outs counted as one hour each. The results were obtained on Intel Xeon CPUs E3-1245 @ 3.40 GHz with 32 GB RAM. It can be seen that SCIPJ/grb is roughly 17 times faster than SPDP, and 96 times faster than Gurobi. For larger instances of the PACE 2018 benchmark, one commonly observes a run time difference of more than six orders of magnitude between SCIP-Jack and commercial MILP solvers.

Considerable problem-specific improvements have been made for the prize-collecting Steiner tree problem (STP) and (to a lesser extent) for the maximum-weight connect subgraph problem [50, 52]. SCIP-Jack 2.0 can solve many previously unsolved benchmark instances from both problem classes to optimality—the largest of these instances have up to 10 million edges. Large improvements are observed for the Euclidean STP: SCIP-Jack 2.0 is able to solve 19 Euclidean STPs with up to 100 000 terminals to optimality for the first time [51]. Notably, the state-of-the-art Euclidean STP solver GeoSteiner 5.1 [30] could not solve any of these instances, even after one week of computation. In contrast, SCIP-Jack 2.0 solves all of them within 12 minutes, some even within two minutes.

[4]See also https://scipjack.zib.de.

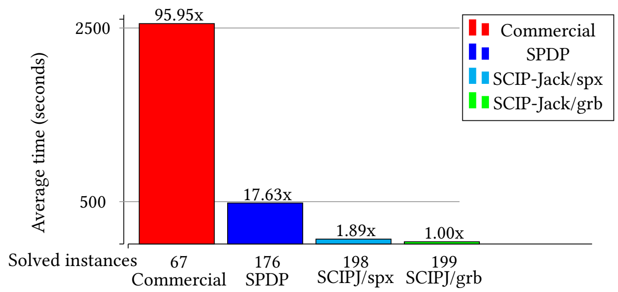

Fig. 2. Computational results on the 200 benchmark instances of Tracks A and B of the PACE Challenge 2018.

5.4 The UG Framework

UG is a generic framework for parallelizing solvers in a distributed or shared memory computing environment. It was designed to parallelize state-of-the-art branch-and-bound solvers externally in order to exploit their powerful performance. We have developed parallel solvers for SCIP [58, 59, 61], CPLEX (not developed anymore), FICO Xpress [60], PIPS-SBB [44, 45], Concorde[5], and QapNB [18]. Customized SCIP-based solvers such as SCIP-SDP and SCIP-Jack can be parallelized with minimal effort [62]. The parallel version of SCIP-Jack solved several previously unsolved instances from SteinLib [35] by using up to 43,000 cores [63].

In addition to the parallelization of these branch-and-bound base solvers, UG was used to develop MAP-SVP [65], which is a solver for the Shortest Vector Problem (SVP), whose algorithm does not rely on branch-and-bound. For these applications, UG had to be adapted and modified. Especially, the success of MAP-SVP, which updated several records of the SVP challenge[6], motivated us to develop *generalized UG*, in which all solvers developed so far can be handled by a single unified framework. This has enabled UG 1.0 to serve as the basis for the parallel frameworks CMAP-LAP (Configurable Massively Parallel solver framework for LAttice Problems) [66] and CMAP-DeepBKZ [67].

6 FINAL REMARKS

We discussed the functionality that the SCIP Optimization Suite offers optimization researchers, and highlighted performance improvements and new functionality that was introduced in the SCIP Optimization Suite 8.0. The performance comparison of SCIP 7.0 and SCIP 8.0 showed a 17 % speed-up on both the MILP and MINLP testsets. This was followed by a discussion of some aspects of the core solving engine of the SCIP Optimization Suite. The new framework for handling nonlinear constraints was presented, which offers increased reliability as well as improved handling of different types of nonlinearities that reduces type ambiguity and extends support for implementing the handling of user-defined nonlinearities. The use of SCIP's flexible plugin-based structure for extending the solver with user methods was demonstrated on the examples of new symmetry handling methods and primal decomposition heuristics. The framework that SCIP provides for working on these methods was explained and the relevant plugin types and other customization-enabling features were discussed, followed by the presentation of new methods added in SCIP 8.0.

Further, we presented extensions built around SCIP. The semidefinite programming solver SCIP-SDP and the Steiner tree problem solver SCIP-Jack provide users of the SCIP Optimization Suite the functionality for solving more problem classes, the decomposition solver GCG offers a different solving approach, and the solver parallelization framework UG

[5] https://www.math.uwaterloo.ca/tsp/concorde.html
[6] http://latticechallenge.org/svp-challenge

enables the use of branch-and-bound solvers, and in particular SCIP, in parallel computing environments. Moreover, these components of the SCIP Optimization Suite demonstrate how SCIP's features can be leveraged in creating new research projects which can extend beyond SCIP's standard focus and approach.

ACKNOWLEDGMENTS

The authors want to thank all previous developers and contributors to the SCIP Optimization Suite and all users that reported bugs and often also helped reproducing and fixing the bugs. In particular, thanks go to Suresh Bolusani, Didier Chételat, Gregor Hendel, Gioni Mexi, Matthias Miltenberger, Andreas Schmitt, Robert Schwarz, Helena Völker, Matthias Walter, and Antoine Prouvoust and the Ecole team. The Matlab interface was set up with the big help of Nicolai Simon.

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), the German Research Foundation (DFG) within the project HPO-NAVI (project number 391087700).

Author Affiliations

KSENIA BESTUZHEVA, Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany; e-mail: bestuzheva@zib.de; ORCID: 0000-0002-7018-7099

MATHIEU BESANÇON, ZIB-AIS^{2}T; e-mail: besancon@zib.de; ORCID: 0000-0002-6284-3033

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

ANTONIA CHMIELA, ZIB-AIS^{2}T; e-mail: chmiela@zib.de; ORCID: 0000-0002-4809-2958

TIM DONKIEWICZ, RWTH Aachen University, Lehrstuhl für Operations Research, Kackertstr. 7, 52072 Aachen, Germany; e-mail: tim.donkiewicz@rwth-aachen.de; ORCID: 0000-0002-5721-3563

JASPER VAN DOORNMALEN, Eindhoven University of Technology, Department of Mathematics and Computer Science; e-mail: m.j.v.doornmalen@tue.nl; ORCID: 0000-0002-2494-0705

LEON EIFLER, ZIB-AIS^{2}T; e-mail: eifler@zib.de; ORCID: 0000-0003-0245-9344

OLIVER GAUL, RWTH Aachen University, Lehrstuhl für Operations Research, Kackertstr. 7, 52072 Aachen, Germany; e-mail: oliver.gaul@rwth-aachen.de; ORCID: 0000-0002-2131-1911

GERALD GAMRATH, ZIB-AIS^{2}T and I^{2}DAMO GmbH, Englerallee 19, 14195 Berlin, Germany; e-mail: gamrath@zib.de; ORCID: 0000-0001-6141-5937

AMBROS GLEIXNER, ZIB-AIS^{2}T and HTW Berlin; e-mail: gleixner@zib.de; ORCID: 0000-0003-0391-5903

LEONA GOTTWALD, ZIB-AIS^{2}T; e-mail: gottwald@zib.de; ORCID: 0000-0002-8894-5011

CHRISTOPH GRACZYK, ZIB-AIS^{2}T; e-mail: graczyk@zib.de; ORCID: 0000-0001-8990-9912

KATRIN HALBIG, Friedrich-Alexander Universität Erlangen-Nürnberg, Department of Data Science, Cauerstr. 11, 91058 Erlangen, Germany; e-mail: katrin.halbig@fau.de; ORCID: 0000-0002-8730-3447

ALEXANDER HOEN, ZIB-AIS^{2}T; e-mail: hoen@zib.de; ORCID: 0000-0003-1065-1651

CHRISTOPHER HOJNY, Eindhoven University of Technology, Department of Mathematics and Computer Science; e-mail: c.hojny@tue.nl; ORCID: 0000-0002-5324-8996

ROLF VAN DER HULST, University of Twente, Department of Discrete Mathematics and Mathematical Programming, P.O. Box 217, 7500 AE Enschede, The Netherlands; e-mail: r.p.vanderhulst@utwente.nl

THORSTEN KOCH, Technische Universität Berlin, Chair of Software and Algorithms for Discrete Optimization, Straße des 17. Juni 135, 10623 Berlin, Germany, and ZIB, Department A^{2}IM, Takustr. 7, 14195 Berlin, Germany; e-mail: koch@zib.de; ORCID: 0000-0002-1967-0077
MARCO LÜBBECKE, RWTH Aachen-Lehrstuhl für OR; e-mail: marco.luebbecke@rwth-aachen.de; ORCID: 0000-0002-2635-0522
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; ORCID: 0000-0002-0499-1820
ERIK MÜHMER, RWTH Aachen-Lehrstuhl für OR; e-mail: erik.muehmer@rwth-aachen.de; ORCID: 0000-0003-1114-3800
BENJAMIN MÜLLER, ZIB-AIS^{2}T; e-mail: benjamin.mueller@zib.de; ORCID: 0000-0002-4463-2873
MARC E. PFETSCH, TU Darmstadt, Fachbereich Mathematik; e-mail: pfetsch@mathematik.tu-darmstadt.de; ORCID: 0000-0002-0947-7193
DANIEL REHFELDT, ZIB-A^{2}IM; e-mail: rehfeldt@zib.de; ORCID: 0000-0002-2877-074X
STEFFAN SCHLEIN, RWTH Aachen-Lehrstuhl für OR; e-mail: steffan.schlein@rwth-aachen.de
FRANZISKA SCHLÖSSER, ZIB-AIS^{2}T; e-mail: schloesser@zib.de
FELIPE SERRANO, ZIB-AIS^{2}T; e-mail: serrano@zib.de; ORCID: 0000-0002-7892-3951
YUJI SHINANO, ZIB-A^{2}IM; e-mail: shinano@zib.de; ORCID: 0000-0002-2902-882X
BORO SOFRANAC, ZIB-AIS^{2}T and TU Berlin; e-mail: sofranac@zib.de; ORCID: 0000-0003-2252-9469
MARK TURNER, ZIB-A^{2}IM and Chair of Software and Algorithms for Discrete Optimization, Institute of Mathematics, TU Berlin; e-mail: turner@zib.de; ORCID: 0000-0001-7270-1496
STEFAN VIGERSKE, GAMS Software GmbH, c/o ZIB-AIS^{2}T; e-mail: svigerske@gams.com
FABIAN WEGSCHEIDER, ZIB-AIS^{2}T; e-mail: wegscheider@zib.de
PHILIPP WELLNER; e-mail: p.we@fu-berlin.de
DIETER WENINGER, Friedrich-Alexander Universität Erlangen-Nürnberg, Department of Data Science, Cauerstr. 11, 91058 Erlangen, Germany; e-mail: dieter.weninger@fau.de; ORCID: 0000-0002-1333-8591
JAKOB WITZIG, ZIB-AIS^{2}T; e-mail: witzig@zib.de; ORCID: 0000-0003-2698-0767

REFERENCES

[1] T. Achterberg. 2007. *Constraint Integer Programming*. Dissertation. Technische Universität Berlin.
[2] Tobias Achterberg. 2009. SCIP: Solving Constraint Integer Programs. *Mathematical Programming Computation* 1, 1 (2009), 1–41. https://doi.org/10.1007/s12532-008-0001-1
[3] Tobias Achterberg, Robert E. Bixby, Zonghao Gu, Edward Rothberg, and Dieter Weninger. 2020. Presolve Reductions in Mixed Integer Programming. *INFORMS Journal on Computing* 32, 2 (2020), 473–506. https://doi.org/10.1287/ijoc.2018.0857
[4] Warren P Adams and Hanif D Sherali. 1986. A tight linearization and an algorithm for zero-one quadratic programming problems. *Management Science* 32, 10 (1986), 1274–1290. https://doi.org/10.1287/mnsc.32.10.1274
[5] Warren P Adams and Hanif D Sherali. 1990. Linearization strategies for a class of zero-one mixed integer programming problems. *Operations Research* 38, 2 (1990), 217–226. https://doi.org/10.1287/opre.38.2.217
[6] Warren P Adams and Hanif D Sherali. 1993. Mixed-integer bilinear programming problems. *Mathematical Programming* 59, 1 (1993), 279–305. https://doi.org/10.1007/BF01581249
[7] Daniel Anderson, Pierre Le Bodic, and Kerri Morgan. 2021. Further results on an abstract model for branching and its application to mixed integer programming. *Mathematical Programming* 190, 1 (2021), 811–841.
[8] Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. 2021. *The SCIP Optimization Suite 8.0.* Technical Report. Optimization Online. http://www.optimization-online.org/DB_HTML/2021/12/8728.html
[9] Víctor Blanco, Alberto Japón, Diego Ponce, and Justo Puerto. 2021. On the multisource hyperplanes location problem to fitting set of points. *Computers & Operations Research* 128 (2021), 105124.

[10] Édouard Bonnet and Florian Sikora. 2018. The PACE 2018 Parameterized Algorithms and Computational Experiments Challenge: The Third Iteration. In *13th International Symposium on Parameterized and Exact Computation, IPEC 2018 (LIPIcs, Vol. 115)*, Christophe Paul and Michal Pilipczuk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 26:1–26:15. https://doi.org/10.4230/LIPIcs.IPEC.2018.26

[11] Marco Casazza and Alberto Ceselli. 2021. Optimization algorithms for resilient path selection in networks. *Computers & Operations Research* 128 (2021), 105191.

[12] Türkü Özlüm Çelik, Asgar Jamneshan, Guido Montúfar, Bernd Sturmfels, and Lorenzo Venturello. 2021. Wasserstein distance to independence models. *Journal of Symbolic Computation* 104 (2021), 855–873.

[13] Alberto Ceselli, Ángel Felipe, M Teresa Ortuño, Giovanni Righini, and Gregorio Tirado. 2021. A branch-and-cut-and-price algorithm for the electric vehicle routing problem with multiple technologies. In *Operations Research Forum*, Vol. 2. Springer, 1–33.

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

[15] Stefano Coniglio, Mathias Sirvent, and Martin Weibelzahl. 2021. Airport capacity extension, fleet investment, and optimal aircraft scheduling in a multilevel market model: quantifying the costs of imperfect markets. *OR Spectrum* 43, 2 (2021), 367–408.

[16] Gustavo Dias and Leo Liberti. 2021. Exploiting symmetries in mathematical programming via orbital independence. *Annals of Operations Research* 298, 1 (2021), 149–182.

[17] Antonio Florez-Gutierrez, Gaëtan Leurent, María Naya-Plasencia, Léo Perrin, André Schrottenloher, and Ferdinand Sibleyras. 2021. Internal Symmetries and Linear Properties: Full-permutation Distinguishers and Improved Collisions on Gimli. *Journal of Cryptology* 34, 4 (2021), 1–37.

[18] Koichi Fujii, Naoki Ito, Sunyoung Kim, Masakazu Kojima, Yuji Shinano, and Kim-Chuan Toh. 2021. *Solving Challenging Large Scale QAPs*. ZIB-Report 21-02. Zuse Institute Berlin.

[19] Tristan Gally. 2019. *Computational Mixed-Integer Semidefinite Programming*. Dissertation. TU Darmstadt.

[20] Tristan Gally, Marc E. Pfetsch, and Stefan Ulbrich. 2018. A Framework for Solving Mixed-Integer Semidefinite Programs. *Optimization Methods and Software* 33, 3 (2018), 594–632. https://doi.org/10.1080/10556788.2017.1322081

[21] 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, and Jakob Witzig. 2020. *The SCIP Optimization Suite 7.0*. Technical Report. Optimization Online. http://www.optimization-online.org/DB_HTML/2020/03/7705.html.

[22] Gerald Gamrath, Thorsten Koch, Stephen J. Maher, Daniel Rehfeldt, and Yuji Shinano. 2017. SCIP-Jack—a solver for STP and variants with parallelization extensions. *Mathematical Programming Computation* 9, 2 (2017), 231–296. https://doi.org/10.1007/s12532-016-0114-x

[23] Gerald Gamrath and Marco E. Lübbecke. 2010. Experiments with a Generic Dantzig-Wolfe Decomposition for Integer Programs. In *Experimental Algorithms (Lecture Notes in Computer Science, Vol. 6049)*, Paola Festa (Ed.). Springer Berlin Heidelberg, 239–252. https://doi.org/10.1007/978-3-642-13193-6_21

[24] Oliver Gaul. 2021. *Hierarchical Strong Branching and Other Strong Branching-Based Branching Candidate Selection Heuristics in Branch-and-Price*. Master's thesis. RWTH Aachen University.

[25] Björn Geißler, Antonio Morsi, Lars Schewe, and Martin Schmidt. 2017. Penalty Alternating Direction Methods for Mixed-Integer Optimization: A New View on Feasibility Pumps. *SIAM Journal on Optimization* 27, 3 (2017), 1611–1636. https://doi.org/10.1137/16M1069687

[26] Ambros Gleixner, Michael Bastubbe, Leon Eifler, Tristan Gally, Gerald Gamrath, Robert Lion Gottwald, Gregor Hendel, Christopher Hojny, Thorsten Koch, Marco E. Lübbecke, Stephen J. Maher, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Franziska Schlösser, Christoph Schubert, Felipe Serrano, Yuji Shinano, Jan Merlin Viernickel, Matthias Walter, Fabian Wegscheider, Jonas T. Witt, and Jakob Witzig. 2018. *The SCIP Optimization Suite 6.0*. Technical Report. Optimization Online. http://www.optimization-online.org/DB_HTML/2018/07/6692.html

[27] A. Gleixner, G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P.M. Christophel, K. Jarck, Th. Koch, J. Linderoth, M. Lübbecke, H.D. Mittelmann, D. Ozyurt, T.K. Ralphs, D. Salvagnin, and Y. Shinano. 2021. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. *Mathematical Programming Computation* 13 (2021), 443–490. https://doi.org/10.1007/s12532-020-00194-3

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

[29] Yoichi Iwata and Takuto Shigemura. 2019. Separator-based pruned dynamic programming for Steiner tree. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 1520–1527.

[30] Daniel Juhl, David M Warme, Pawel Winter, and Martin Zachariasen. 2018. The GeoSteiner software package for computing Steiner trees in the plane: an updated computational study. *Mathematical Programming Computation* 10, 4 (2018), 487–532. https://doi.org/10.1007/s12532-018-0135-8

[31] Volker Kaibel and Andreas Loos. 2011. Finding Descriptions of Polytopes via Extended Formulations and Liftings. In *Progress in Combinatorial Optimization*, A. Ridha Mahjoub (Ed.). Wiley.

[32] Rohit Kannan and James R Luedtke. 2021. A stochastic approximation method for approximating the efficient frontier of chance-constrained nonlinear programs. *Mathematical Programming Computation* 13, 4 (2021), 705–751.

[33] R. Karp. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, R. Miller and J. Thatcher (Eds.). Plenum Press, 85–103. https://doi.org/10.1007/978-1-4684-2001-2_9

[34] Thorsten Koch. 2004. *Rapid Mathematical Prototyping*. Dissertation. Technische Universität Berlin.

[35] Thorsten Koch, Alexander Martin, and Stefan Voß. 2001. *SteinLib: An Updated Library on Steiner Tree Problems in Graphs*. Springer US, Boston, MA, 285–325. https://doi.org/10.1007/978-1-4613-0255-1_9

[36] Leo Liberti and James Ostrowski. 2014. Stabilizer-based symmetry breaking constraints for mathematical programs. *Journal of Global Optimization* 60 (2014), 183–194. https://doi.org/10.1007/s10898-013-0106-6

[37] Jeffrey T Linderoth and Ted K Ralphs. 2005. Noncommercial software for mixed-integer linear programming. *Integer programming: theory and practice* 3 (2005), 253–303.

[38] Stephen Maher, Matthias Miltenberger, João Pedro Pedroso, Daniel Rehfeldt, Robert Schwarz, and Felipe Serrano. 2016. PySCIPOpt: Mathematical Programming in Python with the SCIP Optimization Suite. In *Mathematical Software – ICMS 2016*. Springer, 301–307. https://doi.org/10.1007/978-3-319-42432-3_37

[39] François Margot. 2002. Pruning by isomorphism in branch-and-cut. *Mathematical Programming* 94, 1 (2002), 71–90. https://doi.org/10.1007/s10107-002-0358-2

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

[41] François Margot. 2010. Symmetry in Integer Linear Programming. In *50 Years of Integer Programming*, Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey (Eds.). Springer, 647–686. https://doi.org/10.1007/978-3-540-68279-0_17

[42] Sonja Mars. 2013. *Mixed-Integer Semidefinite Programming with an Application to Truss Topology Design*. Dissertation. FAU Erlangen-Nürnberg.

[43] Frederic Matter and Marc E. Pfetsch. 2021. *Presolving for Mixed-Integer Semidefinite Optimization*. Technical Report. Optimization Online. http://www.optimization-online.org/DB_HTML/2021/10/8614.html

[44] Lluís-Miquel Munguía, Geoffrey Oxberry, and Deepak Rajan. 2016. PIPS-SBB: A Parallel Distributed-Memory Branch-and-Bound Algorithm for Stochastic Mixed-Integer Programs. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 730–739. https://doi.org/10.1109/IPDPSW.2016.159

[45] Lluís-Miquel Munguía, Geoffrey Oxberry, Deepak Rajan, and Yuji Shinano. 2019. Parallel PIPS-SBB: multi-level parallelism for stochastic mixed-integer programs. *Computational Optimization and Applications* 73, 2 (2019), 575–601. https://doi.org/10.1007/s10589-019-00074-0

[46] James Ostrowski, Jeff Linderoth, Fabrizio Rossi, and Stefano Smriglio. 2011. Orbital branching. *Mathematical Programming* 126, 1 (2011), 147–178. https://doi.org/10.1007/s10107-009-0273-x

[47] Tobias Polzin. 2003. *Algorithms For the Steiner Problem in Networks*. Dissertation. Saarland University.

[48] Antoine Prouvost, Justin Dumouchelle, Maxime Gasse, Didier Chételat, and Andrea Lodi. 2021. Ecole: A library for learning inside MILP solvers. *arXiv preprint arXiv:2104.02828* (2021).

[49] Daniel Rehfeldt. 2021. *Faster Algorithms for Steiner Tree and Related Problems: From Theory to Practice*. Dissertation. Technische Universität Berlin.

[50] Daniel Rehfeldt, Henriette Franz, and Thorsten Koch. 2020. *Optimal Connected Subgraphs: Formulations and Algorithms*. ZIB-Report 20-23. Zuse Institute Berlin.

[51] Daniel Rehfeldt and Thorsten Koch. 2021. Implications, conflicts, and reductions for Steiner trees. *Mathematical Programming* (2021). https://doi.org/10.1007/s10107-021-01757-5 To appear.

[52] Daniel Rehfeldt and Thorsten Koch. 2021. On the Exact Solution of Prize-Collecting Steiner Tree Problems. *INFORMS Journal on Computing* (2021). https://doi.org/10.1287/ijoc.2021.1087 To appear.

[53] D. M. Ryan and B. A. Foster. 1981. An integer programming approach to scheduling. In *Computer Scheduling of Public Transport Urban Passenger Vehicle and Crew Scheduling*, A. Wren (Ed.). North Holland, Amsterdam, 269–280.

[54] Domenico Salvagnin. 2018. Symmetry Breaking Inequalities from the Schreier-Sims Table. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, Willem-Jan van Hoeve (Ed.). Springer, 521–529. https://doi.org/10.1007/978-3-319-93031-2_37

[55] Lars Schewe, Martin Schmidt, and Dieter Weninger. 2020. A Decomposition Heuristic for Mixed-Integer Supply chain Problems. *Operations Research Letters* 48, 3 (2020), 225–232. https://doi.org/10.1016/j.orl.2020.02.006

[56] H. D. Sherali and B. M. Fraticelli. 2002. Enhancing RLT relaxations via a new class of semidefinite cuts. *Journal of Global Optimization* 22 (2002), 233–261. https://doi.org/10.1023/A:1013819515732

[57] Yuji Shinano. 2018. The Ubiquity Generator Framework: 7 Years of Progress in Parallelizing Branch-and-Bound. In *Operations Research Proceedings 2017*, Natalia Kliewer, Jan Fabian Ehmke, and Ralf Borndörfer (Eds.). Springer, 143–149. https://doi.org/10.1007/978-3-319-89920-6_20

[58] Yuji Shinano, Tobias Achterberg, Timo Berthold, Stefan Heinz, and Thorsten Koch. 2012. ParaSCIP: A Parallel Extension of SCIP. In *Competence in High Performance Computing 2010*, Christian Bischof, Heinz-Gerd Hegering, Wolfgang E. Nagel, and Gabriel Wittum (Eds.). Springer, 135–148. https://doi.org/10.1007/978-3-642-24025-6_12

[59] Yuji Shinano, Tobias Achterberg, Timo Berthold, Stefan Heinz, Thorsten Koch, and Michael Winkler. 2016. Solving Open MIP Instances with ParaSCIP on Supercomputers Using up to 80,000 Cores. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 770–779. https://doi.org/10.1109/IPDPS.2016.56

[60] Yuji Shinano, Timo Berthold, and Stefan Heinz. 2018. ParaXpress: an experimental extension of the FICO Xpress-Optimizer to solve hard MIPs on supercomputers. *Optimization Methods and Software* 33, 3 (2018), 530–539. https://doi.org/10.1080/10556788.2018.1428602

[61] Yuji Shinano, Stefan Heinz, Stefan Vigerske, and Michael Winkler. 2018. FiberSCIP: A Shared Memory Parallelization of SCIP. *INFORMS Journal on Computing* 30, 1 (2018), 11–30. https://doi.org/10.1287/ijoc.2017.0762

[62] Yuji Shinano, Daniel Rehfeldt, and Tristan Gally. 2019. An Easy Way to Build Parallel State-of-the-art Combinatorial Optimization Problem Solvers: A Computational Study on Solving Steiner Tree Problems and Mixed Integer Semidefinite Programs by using ug[SCIP-*,*]-Libraries. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 530–541. https://doi.org/10.1109/IPDPSW.2019.00095

[63] Yuji Shinano, Daniel Rehfeldt, and Thorsten Koch. 2019. Building Optimal Steiner Trees on Supercomputers by Using up to 43,000 Cores. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, Louis-Martin Rousseau and Kostas Stergiou (Eds.). Springer, Cham, 529–539.

[64] E.M.B. Smith and C.C. Pantelides. 1999. A symbolic reformulation/spatial branch-and-bound algorithm for the global optimisation of nonconvex MINLPs. *Computers & Chemical Engineering* 23, 4-5 (1999), 457–478. https://doi.org/10.1016/s0098-1354(98)00286-5

[65] Nariaki Tateiwa, Yuji Shinano, Satoshi Nakamura, Akihiro Yoshida, Shizuo Kaji, Masaya Yasuda, and Katsuki Fujisawa. 2020. Massive Parallelization for Finding Shortest Lattice Vectors Based on Ubiquity Generator Framework. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41405.2020.00064

[66] Nariaki Tateiwa, Yuji Shinano, Keiichiro Yamamura, Akihiro Yoshida, Shizuo Kaji, Masaya Yasuda, and Katsuki Fujisawa. 2021. *CMAP-LAP: Configurable Massively Parallel Solver for Lattice Problems*. ZIB-Report 21-16. Zuse Institute Berlin.

[67] Nariaki Tateiwa, Yuji Shinano, Masaya Yasuda, Keiichiro Yamamura, Shizuo Kaji, and Katsuki Fujisawa. 2021. *Massively parallel sharing lattice basis reduction*. ZIB-Report 21-38. Zuse Institute Berlin.

[68] Siavash Vahdati Daneshmand. 2004. *Algorithmic Approaches to the Steiner Problem in Networks*. Dissertation. Universität Mannheim.

[69] F. Vanderbeck. 2011. Branching in Branch-and-Price: A Generic Scheme. *Mathematical Programming* 130, 2 (2011), 249–294. https://doi.org/10.1007/s10107-009-0334-1

[70] Stefan Vigerske and Ambros Gleixner. 2017. SCIP: Global Optimization of Mixed-Integer Nonlinear Programs in a Branch-and-Cut Framework. *Optimization Methods & Software* 33, 3 (2017), 563–593. https://doi.org/10.1080/10556788.2017.1335312

[71] D. Villeneuve, J. Desrosiers, M.E. Lübbecke, and F. Soumis. 2005. On Compact Formulations for Integer Programs Solved by Column Generation. *Annals of Operations Research* 139, 1 (2005), 375–388. https://doi.org/10.1007/s10479-005-3455-9

[72] Fabian Wegscheider. 2019. *Exploiting Symmetry in Mixed-Integer Nonlinear Programming*. Master's thesis. Zuse Institute Berlin.

[73] Roland Wunderling. 1996. *Paralleler und Objektorientierter Simplex-Algorithmus*. Dissertation. Technische Universität Berlin.