GERALD GAMRATH, TOBIAS FISCHER, TRISTAN GALLY, AMBROS M. GLEIXNER, GREGOR HENDEL, THORSTEN KOCH, STEPHEN J. MAHER, MATTHIAS MILTENBERGER, BENJAMIN MÜLLER, MARC E. PFETSCH, CHRISTIAN PUCHERT, DANIEL REHFELDT, SEBASTIAN SCHENKER, ROBERT SCHWARZ, FELIPE SERRANO, YUJI SHINANO, STEFAN VIGERSKE, DIETER WENINGER, MICHAEL WINKLER, JONAS T. WITT, JAKOB WITZIG

The SCIP Optimization Suite 3.2

ZIB Report 15-60 (February 2016)

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 3.2

Gerald Gamrath[1], Tobias Fischer[2], Tristan Gally[2], Ambros M. Gleixner[1], Gregor Hendel[1], Thorsten Koch[1], Stephen J. Maher[1], Matthias Miltenberger[1], Benjamin Müller[1], Marc E. Pfetsch[2], Christian Puchert[3], Daniel Rehfeldt[1], Sebastian Schenker[1], Robert Schwarz[1], Felipe Serrano[1], Yuji Shinano[1], Stefan Vigerske[5], Dieter Weninger[4], Michael Winkler[6], Jonas T. Witt[3], and Jakob Witzig[1]

[1]Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany, {gamrath,gleixner,hendel,koch,maher,miltenberger,benjamin.mueller, rehfeldt,schenker,schwarz,serrano,shinano,witzig}@zib.de

[2]Technische Universität Darmstadt, Dolivostr. 15, 64293 Darmstadt, Germany, {tfischer,gally,pfetsch}@mathematik.tu-darmstadt.de

[3]RWTH Aachen University, Kackertstr. 7, 52072 Aachen, Germany, {puchert,witt}@or.rwth-aachen.de

[4]Universität Erlangen-Nürnberg, Cauerstr. 11, 91058 Erlangen, Germany, dieter.weninger@math.uni-erlangen.de

[5]GAMS Software GmbH, c/o ZIB, Takustr. 7, 14195 Berlin, Germany, svigerske@gams.com

[6]Gurobi GmbH, c/o ZIB, Takustr. 7, 14195 Berlin, Germany, winkler@gurobi.com

February 29, 2016

Abstract. The SCIP Optimization Suite is a software toolbox for generating and solving various classes of mathematical optimization problems. Its major components are the modeling language ZIMPL, the linear programming solver SoPlex, the constraint integer programming framework and mixed-integer linear and nonlinear programming solver SCIP, the UG framework for parallelization of branch-and-bound-based solvers, and the generic branch-cut-and-price solver GCG. It has been used in many applications from both academia and industry and is one of the leading non-commercial solvers.
This paper highlights the new features of version 3.2 of the SCIP Optimization Suite. Version 3.2 was released in July 2015. This release comes with new presolving steps, primal heuristics, and branching rules within SCIP. In addition, version 3.2 includes a reoptimization feature and improved handling of quadratic constraints and special ordered sets. SoPlex can now solve LPs exactly over the rational number and performance improvements have been achieved by exploiting sparsity in more situations. UG has been tested successfully on 80,000 cores. A major new feature of UG is the functionality to parallelize a customized SCIP solver. GCG has been enhanced with a new separator, new primal heuristics, and improved column management. Finally, new and improved extensions of SCIP are presented, namely solvers for multi-criteria optimization, Steiner tree problems, and mixed-integer semidefinite programs.

Keywords: mixed-integer linear and nonlinear programming, MIP solver, MINLP solver, linear programming, LP solver, simplex method, modeling, parallel branch-and-bound, branch-cut-and-price framework, generic column generation, Steiner tree solver, multi-criteria optimization, mixed-integer semidefinite programming

Mathematics Subject Classification: 90C05, 90C10, 90C11, 90C30, 90C90, 65Y05

1 Introduction

The SCIP Optimization Suite is a software toolbox for generating and solving mathematical optimization problems. It consists of the modeling language ZIMPL [51, 100], the linear programming solver SoPlex [86, 98], and—as its core—the constraint integer programming and branch-cut-and-price framework SCIP [2, 96]. On top of its modular framework, SCIP provides a full-scale solver for mixed-integer linear programming (MIP) and mixed-integer nonlinear programming (MINLP). Additionally, the SCIP Optimization Suite includes the UG framework [80, 82, 99] for parallelization of branch-and-bound-based solvers and the generic branch-cut-and-price solver GCG [35, 93].

The focus of this report lies on the new features and improvements added in version 3.2 of the SCIP Optimization Suite. This covers the major release 3.2.0 from July 2015 as well as the 3.2.1 bugfix version released February 2016. For the general concepts of the SCIP Optimization Suite and its components we refer to the references given above and the documentation available at the project website `http://scip.zib.de`.

The paper is structured as follows. Each of the next five sections is dedicated to one component of the SCIP Optimization Suite: SCIP (Section 2), SoPlex (Section 3), ZIMPL (Section 4), UG (Section 5), and GCG (Section 6). Section 7 presents three new and improved extensions of SCIP. An evaluation of the performance improvements of the SCIP Optimization Suite for solving MIPs and MINLPs can be found in Section 8. Finally, Section 9 gives an outlook on the future development of the SCIP Optimization Suite.

2 SCIP: Solving Constraint Integer Programs

SCIP 3.2 introduces many new plugins and performance improvements. This release has a particular focus on presolving, primal heuristics, and branching rules for mixed-integer linear programming. In regards to mixed-integer nonlinear programming this release includes improvements to separation, propagation and constraint upgrading processes. Additionally, SCIP now features the functionality for the reoptimization of mixed-binary programs. The following is an overview of all new features of SCIP included in this release. A detailed description of each feature is provided later in this section.

- Reoptimization: solving mixed-binary programs with changed objective function or tighter feasible region (Section 2.1)
- Presolving (Section 2.2):
  - five new presolvers
  - new presolving and propagation algorithm for linear ranged rows and equations using greatest common divisor arguments
  - a global matrix module that provides access to the matrix representation of a mixed-integer linear program
  - improved upgradability of continuous variables to implicit integer variables
  - the introduction of presolving levels to classify presolving steps based on their runtime complexity and improve their coordination
- Primal Heuristics (Section 2.3):
  - the development of new primal heuristics: distribution diving, indicator and bound heuristics
  - improved clique and variable bound heuristics
  - revision of the diving heuristics implementation: constraint handlers can register fixing candidates, dynamic LP solving to solve fewer diving LPs
- Branching Rules (Section 2.4):

  - new branching rules: branching on multi-aggregated variables and distribution branching
  - new reliability notions in hybrid reliability pseudo-cost branching
  - improvement in the treatment of nonlinearities in hybrid reliability pseudo-cost branching

- Mixed-Integer Nonlinear Programming (Section 2.5):
  - a new separator for edge-concave cuts
  - the generalized upgrade from quadratic constraints to SOC constraints
  - an improved separation procedure for convex quadratic constraints
  - improved optimization-based bound tightening by applying separation and propagation
  - support for user-defined operators in expression trees/graphs

- SOS1 constraint handler (Section 2.6):
  - a new branching rule for SOS1 constraints
  - an improved separation procedure
  - registration of variables in SOS1 constraints as candidates for diving heuristics, also discussed in Section 2.3

- Further changes (Section 2.7):
  - an extension of the probing mode to allow separation and objective coefficient changes
  - the transfer of history information to and from sub-SCIPs
  - the use of sparsity information during separation
  - better handling of large values returned by the LP solver
  - an improvement of the memory management
  - the ability to output information for BAK: Branch-and-bound Analysis Kit
  - provided more options for the final feasibility check
  - added the option to permute the original problem directly after reading

2.1 Reoptimization

The new reoptimization feature of SCIP can be used to solve a sequence of mixed-binary programs (P_1), …, (P_N) where

$$(P_i) \qquad \min\{c_i^\top x \,:\, A_i x \geq b_i,\ \ell \leq x \leq u,\ x \in \{0,1\}^k \times \mathbb{R}^{n-k}\}$$

with objective function c_i, constraint matrix A_i, and right-hand side b_i. This sequence is given implicitly, i.e., at each point in time, only the current problem P_i is known and the definition of P_{i+1} may depend on the solution computed for problem P_i. There are no restrictions on the changes in the objective function, but it is required that the feasible region is not extended within the sequence, i.e., the feasible region of P_i contains that of P_{i+1}. The basic idea behind the new feature is to reuse the branch-and-bound tree built during the solving process of P_i for solving P_{i+1}. This idea can be realized in combination with a simple branch-and-bound algorithm without major difficulties and has been previously used by several authors in both general and application-specific approaches, e.g., generating multiple solutions for mixed-integer programs [21], investigations on the value function of mixed-integer programs [72], duality warmstart [71], and in a special branch-and-bound framework for elevator scheduling [47]. In the following, we give a short overview how reoptimization can be used in a state-of-the-art MIP solver in the context of a sophisticated branch-and-bound algorithm, e.g., cope with dual reductions. For a detailed description we refer to [85].

Important ingredients of state-of-the-art MIP solvers are presolving and bound tightening techniques that use primal and dual reductions. A reduction is called *primal* if all feasible solutions are preserved and *dual* if it may remove feasible solutions (but preserves at least one optimal solution to guarantee correctness). Since solutions pruned by dual reductions may become optimal after the objective function is changed, dual reductions are only valid for the current objective function. Hence, those reductions need to be treated with special care to guarantee optimality after changing the objective function. In the current release all feasible solutions pruned by dual reductions are reconstructed as follows: Consider a node v of the search tree and a set of binary variables C fixed at that node by dual reductions. Moreover, let $C^+ := \{i \in C : x_i = 1\}$ and $C^- := \{j \in C : x_j = 0\}$ be a partition of C based on the fixing value. Due to the bound tightenings based on dual information all feasible solutions fulfilling

$$\bigvee_{i \in C^+} (x_i = 0) \vee \bigvee_{j \in C^-} (x_j = 1)$$

are pruned from node v. In the current release all these solutions are reconstructed by adding a copy v' of node v with variables in C unfixed and extended by the constraint

$$\sum_{i \in C^+} (1 - x_i) + \sum_{j \in C^-} x_j \geq 1.$$

Since the duplication of all nodes where dual reductions were performed rapidly enlarges the tree, SCIP 3.2 contains two heuristics that construct a new search tree of smaller size containing as much important information as possible from the original branch-and-bound tree. Moreover, the latest release provides a new primal heuristic that is built specifically for the reoptimization case and modifies an optimal solution of the previous solving process based on the changes in the objective function.

2.2 Presolving

Presolving aims at reducing the size and, more importantly, improving the formulation of a given model. It has been shown that presolving is powerful in practice and a key factor in solving mixed-integer programs [14]. In this section, the new presolving techniques included in SCIP 3.2 are briefly presented. After that, the matrix module is described, which allows a column view on the problem paving the way for many of the new presolving methods. Finally, the coordination of the presolving process was improved by *presolving levels*, which is introduced at the end of the section.

Singleton column stuffing. Consider the binary knapsack problem. The task is to select a set of items, characterized by profits and sizes, such that these items fit into a knapsack of given capacity while maximizing the total profit. In the LP relaxation of this problem, it is allowed to pack any fraction of an item. Thus, an optimal solution to the relaxed problem can be achieved by sorting the items by their profit/size ratio and selecting them in this order until the capacity is reached [23]. The *stuffing* presolver transfers this idea for singleton continuous variables occurring in one common constraint of mixed-integer programs. Details on the theoretical background and implementation can be found in [33].

Redundant variable upper bounds. In many real world problems, e.g., production planning, variable upper bound constraints of the form $x - Cy \leq 0$ are present, with variables $x \in \mathbb{R}_+$, $y \in \{0, 1\}$, and a coefficient $C \geq 0$. In the context of production planning, this can be regarded as follows: If a product should be produced (corresponding to having a strictly positive value for x), the corresponding machine needs to be switched on (represented by the assignment $y = 1$). The *redvub* presolver aims at detecting redundant variable upper bound constraints on the same variable x but different binary variables. For example, if there exists another constraint $x - Dz \leq 0$ with $C \leq D$ and $z \in \{0, 1\}$, this constraint is dominated by

$x - Cy \leq 0$. Therefore, the coefficient D can be tightened to C in the second constraint. In some cases, even $z = y$ can be aggregated and the constraint $x - Dz \leq 0$ can be deleted. For example, this reduction is valid if the objective function coefficients of y and z are both non-positive or both non-negative and both variables are singleton columns. This approach can be transferred to the case of variable lower bounds as well.

Implied free variables. Let ℓ_j and u_j denote the explicit lower and upper bounds of variable x_j stated in the problem formulation. Further let $\bar{\ell}_j$ and $\bar{u}_j$ be the tightest bounds on x_j obtained by activity-based bound tightenings of constraints in the model [17, 75]. If $\bar{\ell}_j$ and $\bar{u}_j$ are at least as tight as the original bounds, i.e., $[\bar{\ell}_j, \bar{u}_j] \subseteq [\ell_j, u_j]$, variable x_j is said to be *implied free* [7]. The *implfree* presolver multi-aggregates continuous implied free variables appearing in equality constraints if specific requirements concerning fill-in and numerical stability are fulfilled. In some cases this approach can even be applied to integer variables.

Two-row bound tightening. The conventional activity-based bound tightening technique considers single constraints. By examining pairs of constraints it is sometimes possible to derive tighter bounds on the variables that appear in either of the two constraints. Assume there are constraints $ax + by \leq c$ and $dx + ez \leq f$ with common variables x. By solving two linear programs $L = \min\{dx : ax + by \leq c\}$ and $U = \max\{dx : ax + by \leq c\}$, i.e., $L \leq dx \leq U$, we obtain tighter activities for the second constraint and may derive tighter bounds on z than single constraint activity-based bound tightening. This presolving approach was first published in [4] and is implemented in the SCIP 3.2 presolver plugin *tworowbnd*.

Dual aggregations. Let the MIP $\min\{c^\top x : Ax \geq b,\ x \in [\ell, u],\ x \in \mathbb{Z}^k \times \mathbb{R}^{n-k}\}$ be given. We assume there exists a variable x_j with $c_j \leq 0$, bounds $[\ell_j, u_j]$, and only constraint i prevents fixing x_j to its upper bound. Further there is a binary variable x_k with a negative coefficient in row i. If by setting $x_k = 0$, constraint i becomes redundant and from setting $x_k = 1$ it follows that x_j is at its lower bound, then the aggregation $x_j := u_j + (\ell_j - u_j) \cdot x_k$ can be applied. This idea is easily transferable to similar cases. More details on this presolving technique can be found in the source code of the presolver plugin *dualagg*.

Using greatest common divisors in ranged row propagation. Given a linear constraint $\underline{b} \leq \sum_{i \in I} a_i x_i \leq \bar{b}$, where $\underline{b}$ can—but does not need to—be equal to $\bar{b}$. The set of variables is partitioned into two sets I_1, I_2 with the following properties: I_1 contains only integral variables and there exists $g > 1$ such that the greatest common divisor (gcd) of all pairs of coefficients $a_i, a_j, i, j \in I_1$ is a multiple of g. Let g be the largest value for which the previous condition holds for the given set I_1. The set I_2 contains all remaining variables. Consider the case where both sets are non-empty. Using the gcd g and activity information on both sets, it is possible to detect infeasibility, bound changes, fixings, or tighter sub-constraints can be identified. For example, let m_2 and M_2 be the minimum and maximum activity of set I_2, respectively. If

$$\begin{aligned}
&\min\{m_2 + \alpha \cdot g : \alpha \in \mathbb{Z},\ m_2 + \alpha \cdot g \geq \underline{b}\} > \bar{b} \quad \text{and} \\
&\max\{M_2 + \beta \cdot g : \beta \in \mathbb{Z},\ M_2 + \beta \cdot g \leq \bar{b}\} < \underline{b},
\end{aligned}$$

there is no feasible solution for this constraint and infeasibility of the current problem is proven.

Upgrade to implicit integer variables. The existing upgrade routine for continuous variables in linear equations has been extended. The new implementation also handles the case of an equation with integral coefficients, integral right-hand side, and all but one variable being of integral type. The continuous variable x_i with coefficient a_i in the constraint is replaced by an implicit integer variable z with coefficient 1. They are connected via the implicit aggregation equality $x_i = z/a_i$.

Matrix module. As a framework for constraint integer programming, SCIP allows to define various constraint types by implementing constraint handler plugins. This approach is very flexible and allows to solve a variety of problem classes, but implies that the problem can only be accessed in a constraint-based fashion. Sometimes it is also useful to have a column view. SCIP 3.2 has been extended to provide the ability to generate the matrix given by the variables and all constraints of linear type and specializations, i.e., all `linear`, `set partitioning/packing/covering`, `knapsack`, `logic or`, and `variable bound` constraints. This matrix is internally represented in *row* and *column major format*, i.e., as ordered and contiguous arrays of nonzero entries of the rows and columns, respectively [16].

SCIP 3.2 offers some functions for having access to the row and column major format. In this section, only a selection of useful functions for the column major case is described. The complete list of functions can be found in `src/scip/pub_matrix.h`. Before the matrix can be used, it needs to be created by the method `SCIPmatrixCreate()`. The number of non-zeros of a column can then be accessed with `SCIPmatrixGetColNNonzs()`, the nonzero entries and their row indices can be accessed with `SCIPmatrixGetColValPtr()` and `SCIPmatrixGetColIdxPtr()` respectively. In the end, the memory of the matrix module needs to be freed by calling `SCIPmatrixFree()`.

Presolving levels. In order to improve the coordination of the increasing number of presolvers, SCIP 3.2 introduces the concept of presolving levels. Presolving steps are now classified as `fast`, `medium`, or `exhaustive` algorithms. While global presolving plugins typically implement one presolving step associated to a specific presolving level, constraint-type specific presolving callbacks implemented in the constraint handlers typically cover a set of presolving reductions of different complexity. Therefore, a presolving callback can be called in several presolving levels and decides, based on the current level, which of the algorithms should be applied. For example, most of the presolving steps of the linear constraint handler like activity-based bound tightening or coefficient tightening are relatively cheap and are therefore applied in presolving level `fast`. There are, however, very expensive procedures like redundancy detection of constraints with a potentially quadratic effort for pairwise comparisons which are only called in presolving level `exhaustive`.

Each presolving round starts with running the `fast` algorithms and only calls `medium` and `exhaustive` ones, if the reductions obtained by presolving so far are not enough to trigger another presolve round. Thus, a presolving round can consist of only calling all `fast` presolving steps once, if those suffice to obtain the desired reduction rate as specified by parameter `presolving/abortfac`. Otherwise, all presolving algorithms of level `medium` are called and the reduction rate is checked again, either triggering a new presolving round starting with the `fast` presolving methods again or advancing to the `exhaustive` ones. Due to the high complexity of `exhaustive` presolving steps, the new reduction rate is checked after each call of a presolver during the `exhaustive` phase and as soon as the desired rate is reached, a new presolving round is triggered (as opposed to the behavior in the `fast` and `medium` level, where all respective presolvers are executed before checking for the reduction rate). If after executing all `exhaustive` presolving methods, the desired reduction rate is still not reached, presolving is terminated.

The information printed during presolving has been adjusted accordingly. A typical output looks like this:

```
presolving:
(round 1, fast)       2 del vars, 4 del conss, ..., 0 impls, 13 clqs
(round 2, fast)       7 del vars, 11 del conss, ..., 0 impls, 13 clqs
(round 3, exhaustive) 15 del vars, 24 del conss, ..., 2 impls, 10 clqs
(round 4, medium)     19 del vars, 32 del conss, ..., 3 impls, 14 clqs
(round 5, fast)       25 del vars, 47 del conss, ..., 65 impls, 10 clqs
(round 6, exhaustive) 48 del vars, 66 del conss, ..., 70 impls, 4 clqs
presolving (7 rounds: 7 fast, 4 medium, 3 exhaustive):
 48 deleted vars, 66 deleted constraints, ...
 70 implications, 4 cliques
```

In this case, SCIP performed seven rounds of presolving. For each round the maximum presolving level reached in that round is printed as well as the reductions found so far. Note that for the last round, SCIP does not print an extra line, since possible reductions performed during that round can be identified in the summary printed at the end. In the first two rounds the model was sufficiently reduced by `fast` presolving steps only, round 3 called all `fast` and `medium` presolving methods without significant reductions and only an `exhaustive` algorithm was successful enough. Round 4 went up to `medium` presolving and round 5 was successful enough with only `fast` presolving. Finally, round 6 and 7 went up to `exhaustive` presolving. While round 6 was able to significantly reduce the model, no presolver had any success in round 7 and thus presolving was stopped. The summary at the end shows the number of rounds in which presolving of a certain level was performed. Note that since each round starts with `fast` presolving, the number of rounds with `fast` presolving is always equal to the total number of presolving rounds.

2.3 Primal heuristics

The current SCIP release contains three new heuristics as well as improvements to several existing heuristics. In particular, primal diving heuristics have been redesigned and algorithmically changed.

Distribution diving. This new primal heuristic extends the set of primal diving heuristics of SCIP. Diving heuristics explore a single path in an auxiliary search tree with a branching rule that differs from the rule used in the main tree. Distribution diving branches on variables which strongly reduce the estimated *solution density* [67] of the linear constraints in the created subproblems. As solution density of a linear inequality, we denote the fraction of feasible assignments among all possible assignments to the variables occurring in this inequality. The solution density can be approximated by means of a normal distribution. In [69], solution densities are proposed for branching in MIP with the aim to find feasible solutions quickly by forcing changes on other variables. This makes it well-suited for use in a primal diving heuristic.

To estimate the solution density for a linear inequality $a^\top x \leq b$, we formally replace the variables x_j by random variables X_j following a uniform distribution over their domain. We then approximate the probability $\mathbb{P}(a^\top X \leq b)$ by means of a normally distributed variable $\sim \mathcal{N}(\mu, \sigma^2)$ using the mean μ and variance σ^2 of $a^\top X$ using Lyapunov's central limit theorem, see [67] for a proof. In the case of equality constraints, the authors [69] propose the use of a centrality measure that combines the two solution densities of the involved inequalities.

Figure 1 shows the relevant distributions for an example constraint $x + 0.5y + z \leq 10$ that involves three integer variables $x, z \in \{0, \ldots, 5\}$, and $y \in \{0, \ldots, 10\}$. The activity of the constraint is modeled as a random variable Q with mean value $\mu = 7.5$ and variance $\sigma^2 \approx 8.3$. The top picture shows the probabilities that Q takes any particular value $t \in \{0, 0.5, 1, \ldots, 15\}$. It also shows the probability density function $f_{\mu,\sigma^2}(t)$ of a normally distributed variable $R \sim \mathcal{N}(\mu, \sigma^2)$. The second diagram reveals the close relationship between the cumulative distribution functions of Q and R; the actual probability $\mathbb{P}(Q \leq 10) \approx 0.82$ is approximated well by $\Phi_{\mu,\sigma^2}(10) = \mathbb{P}(R \leq 10) \approx 0.81$.

The branching variable selection of the heuristic can be controlled by the user, where the five possible options include the choice of a variable-direction pair that minimizes the solution density of any affected inequality after branching. With default settings, the heuristic cycles through the five different selection methods because experimental results do not reveal any of the strategies to be superior.

For more information and details about the distribution diving heuristic in SCIP, we refer to [44]. Please see also the paragraph below on the general adjustment of diving heuristics.

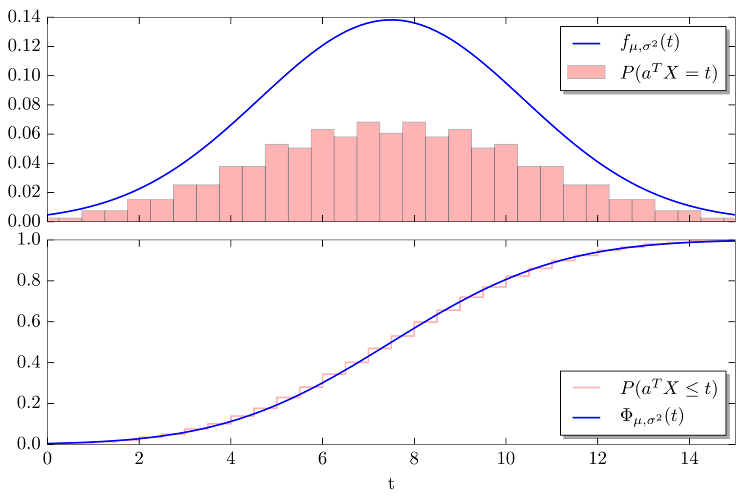

Figure 1: The actual probability density of $Q \coloneqq X + 0.5Y + Z$, where $X, Z \sim \mathcal{U}(\{0, \ldots, 5\})$ and $Y \sim \mathcal{U}(\{0, \ldots, 10\})$, and its approximation by a normal distribution $\mathcal{N}(\mu, \sigma^2)$. The actual solution density is $\mathbb{P}(Q \leq 10) \approx 0.82$, whereas the normal approximation yields 0.81.

Indicator heuristic. Given a binary variable y and some constraint $\mathcal{C}$, an *indicator constraint* is of the following form: $y = 1 \to \mathcal{C}(x)$, i.e., if $y = 1$ then constraint $\mathcal{C}$ has to be fulfilled by variables x. SCIP can handle this general form through the `superindicator` constraint handler. For the special case of a linear constraint $\mathcal{C} = \{x : a^\top x \leq \beta\}$, additional presolving can be performed and cutting planes can be added; here the `indicator` constraint handler is responsible.

In general, an indicator constraint is handled by branching on y. In the $y = 0$ branch, $\mathcal{C}$ is not enforced, while in the $y = 1$ branch, $\mathcal{C}$ is enforced; in the linear case, the corresponding linear constraint is separated for the $y = 1$ branch.

Since an indicator constraint is only indirectly represented in LP relaxations, often a relaxation solution has $y = 1$, but the secondary constraint $\mathcal{C}$ is not fulfilled. Moreover, such solutions can sometimes also be produced by the internal mechanisms of the indicator constraint handler. In the linear case, the indicator heuristic tries to turn such partial solutions into generally feasible solutions. To this end, the binary indicator variables of all indicator constraints are fixed to their values and the linear program including the linear inequalities corresponding to $y = 1$ variables is solved in the hope that this produces a feasible solution. Moreover, one can also perform a one-opt approach, i.e., the values of single binary y-variables are flipped and the corresponding LP is solved.

Both methods are activated and triggered by solutions coming from the indicator constraint handler, if there are indicator constraints present.

Bound heuristic. The bound heuristic is a very simple start heuristic. It first fixes all binary and integer variables to their lower or upper bound and then solves an LP on the remaining problem. A parameter is used to specify whether the variables should all be fixed to their upper bound or their lower bound, or whether two runs should be done, one for each bound. If the bound to which a variable should be fixed is infinite, the heuristic run is terminated.

This heuristic is inspired by production models in which fixing all startup variables to 1 may give a solution that optimizes the continuous production disregarding startup costs, while fixing all startup variables to 0 leads to a solution with 0 production, such that stocked products are optimally distributed and the penalty cost for not fulfilling some of the demands is minimized.

Improved clique and variable bound heuristics. The clique and variable bound heuristics use a large neighborhood search (LNS) approach to generate primal solutions. In contrast to other LNS heuristics for MIP like RINS [22], RENS [12], and Local Branching [27] they do not rely on a feasible or LP solution to define the neighborhood, but rather use two structures globally available in SCIP, the *clique table* and the *variable bound graph*. A clique is a set of binary variables of which at most one can be set to 1, a variable bound is a bound on one variable which depends on the value of another variable. Such structures are identified during presolving and are collected in the clique table and the variable bound graph, respectively. The heuristics can be called before solving the root node LP. They first define the neighborhood by repeatedly fixing a (set of) variables and applying domain propagation on the reduced problem to identify consequences on the bounds of other variables. The clique table and the variable bound graph are used to predict this effect and help to define different fixing orders, e.g., fixing variables first which lead to many other bounds being changed and thus to reaching a feasible solution or conflicting bounds fast. After the fixing process is finished, the LP relaxation of the remaining problem is solved and a simple rounding heuristics is applied to the solution. Only if this procedure did not already generate a feasible solution, the remaining problem is solved as a sub-SCIP. Although the two heuristics were already included in SCIP before, they are significantly improved in SCIP 3.2. Most importantly, the fixing and propagation mechanism was rearranged and several new fixing schemes were introduced. For more details, we refer to [31].

Unification and adjustment of diving heuristics. The primal diving heuristics of SCIP have been redesigned with two goals in mind: First, more general branching decisions are incorporated in order to allow for branching on indicator variables or variables in SOS1 constraints. Second, a performance improvement is obtained by lowering the frequency of nodes at which LPs are solved. As side effect, the code duplication within the implementation of the diving heuristics is significantly reduced.

Primal diving heuristics in SCIP explore a single probing path down the tree with a particular variable selection rule inside an auxiliary search tree. Due to warmstart capabilities of the dual simplex algorithm and the restriction to only explore a single probing path, the diving procedure is usually fast. We distinguish primal diving heuristics from objective diving heuristics, which attempt to attain integral feasible solutions by so-called soft roundings that are induced by temporal changes to the objective coefficient of fractional variables. A description of the existing primal and objective diving heuristics in SCIP can be found in [1, 11, 43]. The new primal heuristic distribution diving was added to version 3.2 of SCIP.

The introduction of a dynamic LP solving frequency constitutes the main change during the execution of the primal diving heuristics. After the bound of a variable is changed, propagation routines are called to deduce further bound tightenings on other variables and reduce the local subproblem in size. The LP, however, is not reoptimized at every diving node, but bound changes and propagation may be iterated until a certain percentage of domain reductions on all variables is found during probing.

The LP resolve behavior of each diving heuristic is controlled by two user parameters: its LP solving frequency `lpsolvefreq` and an LP resolve domain change quotient, denoted by `lpresolvedomchgquot`. While using a positive LP solving frequency turns on LP resolving at a static, positive frequency, setting it to 0 yields the new, dynamic behavior depending on the number of intermediate domain reductions. The second parameter can be adjusted to control the number of domain changes, as a percentage of the problem variables, required to resolve the LP. Note that setting the LP solving frequency to 1 restores the algorithmic behavior of primal diving heuristics prior to SCIP version 3.2.0.

This algorithmic change is accompanied by a redesign of the diving heuristics. Their behavior is now controlled by `diveset` structures, which all primal diving heuristics include. An included `diveset` installs all relevant settings for the diving heuristic plugin and requires the implementation of a variable scoring callback `SCIP_DECL_DIVESETGETSCORE`, see the coefdiving-heuristic for an example. The execution of the heuristic calls the new method

`SCIPperformGenericDivingAlgorithm()`, which takes a `diveset` as argument and subsequently performs diving with the given scoring mechanism. More on the implementation of a primal diving heuristic using the new controller can be found in a dedicated section of the SCIP-documentation.

In order to generalize the diving scheme to incorporate constraints other than just integrality restrictions, SCIP now provides a new, optional constraint handler callback, `SCIP_DECL_CONSGETDIVEBDCHGS`. By implementing this callback, one allows the constraint handler to participate in the diving variable selection. All constraint handlers that implement this callback are called in decreasing order of their enforcement priority, similarly to the way branching is performed during the main search. The constraint handler can use the scoring mechanism of the passed `diveset` and buffer its desired set of dive bound changes.

Furthermore, the statistic output of SCIP now contains a section about diving heuristics. It shows the average, minimal and maximal depth as well as the number of explored probing nodes, backtracks, and used LP iterations for every primal diving heuristic.

2.4 Branching

SCIP 3.2 comes with two new branching rules for MIP and several enhancements of the default hybrid reliability pseudo-cost branching rule for MIP and MINLP. Additionally, branching for problems with SOS1 constraints has been improved, which is discussed in Section 2.6

Distribution branching. Distribution branching is a branching variable selection rule that approximates solution densities in the subtrees created by branching. This idea was first presented in [69] in an effort to quickly find feasible solutions in MIP. Distributed branching has been incorporated it into SCIP as both a branching rule and a diving heuristic. For more information and further references, we refer to the paragraph on the distribution diving primal heuristic in Section 2.3.

Branching on multi-aggregated variables. The SCIP 3.2 release introduces the multi-aggregated branching rule [36], which is the first branching rule in SCIP for general MIPs that branches on general disjunctions. It only branches on disjunctions defined by multi-aggregated variables. These are variables that were removed from the problem during the presolving due to linear dependencies within the variable set. More specifically, a multi-aggregated variable is replaced throughout the problem by an affine linear sum of other variables: $x_k = \sum_{j\in\mathcal{S}} \alpha_j x_j + \beta$, where $\mathcal{S}$ is a subset of the variable set not including x_k. While this procedure reduces the problem size and typically improves the performance of MIP solvers, it also restricts the degree of freedom in variable-based branching rules, since a multi-aggregated variable does not serve as a candidate for branching anymore. The multi-aggregated branching rule attempts to overcome this drawback. It considers not only integer variables x_i with fractional LP solution value $\tilde{x}_i$ as branching candidates, but also checks for all integer multi-aggregated variables if their value in the current LP solution $\sum_{j\in\mathcal{S}} \alpha_j \tilde{x}_j + \beta$, is fractional. In that case, the following general disjunction is added to the set of branching candidates:

$$\sum_{j\in\mathcal{S}} \alpha_j x_j \le \left\lfloor \sum_{j\in\mathcal{S}} \alpha_j \tilde{x}_j \right\rfloor \quad \bigvee \quad \sum_{j\in\mathcal{S}} \alpha_j x_j \ge \left\lfloor \sum_{j\in\mathcal{S}} \alpha_j \tilde{x}_j \right\rfloor + 1.$$

All branching candidates—single variables as well as general disjunctions—are then evaluated with a strong branching approach and the one providing the best dual bound improvement is selected. So far, this branching rule is not competitive for general MIP instances. However, it can be quite effective for special problem types like Scheduling problems, cf. [36]. This is particularly evident if the multi-aggregated variables represent high-level decisions that lead to a more balanced branch-and-bound tree.

New reliability notions in hybrid reliability pseudo-cost branching. The state-of-the-art variable selection scheme for MIPs is called *reliability pseudo-cost branching* [5]. This rule uses estimated dual gains via pseudo-costs [9] based on previous branching observations to rank the branching candidate variables as soon as this information has become *reliable*; it performs strong branching to approximate the dual gain on unreliable candidates and updates the available candidate branching history accordingly. The originally proposed notion of reliability uses a fixed, variable independent threshold η on the number of observations after which variables becomes reliable.

Several implementation tricks are used in SCIP to limit the computational effort for strong branching further, e.g., an iteration limit for every strong branching LP that should not exceed twice the number of average dual simplex iterations per call so far. Furthermore, the fixed reliability threshold is decreased if strong branching iterations are very expensive compared with the number of iterations spent for resolving LPs at the actual search nodes.

The idea for introducing novel notions of reliability is based on the observation that a fixed threshold might not account for all variables equally well. Besides, a fixed threshold that needs to be set by the user might not scale well with an increasing problem size. Together with the objective gain per unit fractionality, which is stored incrementally to be used for the pseudo-cost based prediction, we incrementally calculate the dual gain variance (normalized by the fractionality) for every candidate variable.

With this new information at hand, we introduce *relative-error reliability* and *hypothesis reliability* [45]. The first notion computes confidence intervals around the average unit gains in order to compute relative errors. It marks branching candidates with a large relative error unreliable and therefore continue strong branching on those candidates. The second notion, hypothesis reliability, uses a variant of a one-sided t-test [55] in order to concentrate strong branching on candidates with a gain history insignificantly lower than that of the best pseudo-candidate at a node. This latter notion of reliability is also used to skip strong branching on candidates with a significantly lower score than the current candidate.

Both novel notions of reliability can be activated via the advanced user parameters `usehyptestforreliability` and `userelerrorreliability` under `branching/relpscost`. Upper and lower bounds for the relative error to be reliable can also be specified and are used dynamically depending on the overall strong branching effort. In addition, the confidence level used for the interval construction and hypothesis tests can be adjusted. Whenever activated, the two new notions are used after the classical (fixed-threshold) reliability, i.e., variables with too few observations are considered unreliable anyway. If both are active, variables are considered unreliable only if they are unreliable in terms of both notions.

With the current release, hypothesis reliability is used inside an optimality-based emphasis setting. The dual gain variance of the variables can now be accessed together with the other branching history information from the interactive shell via the command

```
SCIP> display varbranchstatistics[enter]
                     ...      LP gain          pscostcount       gain variance
variable  prio   ...   down          up   down     up   down          up
t_x0001      0   ...  1.1654      0.7234  137.0  138.0  0.62        0.12
t_x0002      0   ...  1.5283      0.7970  208.0  203.0  0.76        0.10
t_x0003      0   ...  1.7695      0.7431   62.0   65.0  0.56        0.15
t_x0004      0   ...  0.8295      0.7050   23.0   53.0  0.40        0.15
t_x0005      0   ...  1.5882      0.7645   29.0   30.0  0.85        0.13
t_x0006      0   ...  0.5447      0.7723   12.0   28.0  0.25        0.2
```

Nonlinearities in hybrid reliability pseudo-cost branching. When solving nonconvex MINLPs the default branching scheme of SCIP is hierarchical. If an integer variable with fractional relaxation value is available, the hybrid reliability pseudo-cost branching rule (`relpscost`) selects one of these variables. Only if the relaxation solution is integer feasible, spatial branching is performed.

With version 3.2, the computation of the hybrid branching score in `relpscost` branching uses information about the "nonlinearity" of a variable: the number of nonlinear terms of the constraint functions in which the variable appears. The motivation behind this concept is that variables appearing in many nonlinear terms should be preferred, because branching on them potentially reduces the violation of nonlinear constraints and may have a positive effect on their enforcement in the subsequent search.

We normalize the nonlinearity of each variable by the maximal nonlinearity of all variables and use this normalized value as a new summand in the hybrid score. Its weight is controlled by the parameter `branching/relpscost/nlscoreweight`, by default equal to 0.1; hence it is less important than pseudo-costs, but more important than conflict and inference values. Note that this only has an effect if the problem contains "nonlinear integer variables". If there are no nonlinear constraints or if all nonlinear constraints are defined over continuous variables the score computation is not affected.

Empirically, we observed a significant reduction in running time by 17.1% on the affected instances of the benchmark set MINLPLib2. Additionally, six more instances are now solved within one hour. For more details see [37].

2.5 Mixed-Integer Nonlinear Programming

Besides the changes in hybrid branching discussed in the previous section, the main improvements for MINLP in SCIP 3.2 are improved separation and upgrade capabilities of the quadratic constraint handler. In addition, optimization-based bound tightening has been enhanced and the expression framework can now be extended by user-defined operators.

New separator for edge-concave cuts. The edge-concave cut separator decomposes a quadratic constraint $x^\top Qx + d^\top x + b \le 0$ into edge-concave terms f_i and a remaining term r by solving an auxiliary MIP. Each f_i is chosen such that it is edge-concave with respect to the domain box $[\ell, u] = \{x \in \mathbb{R}^n : \ell_i \le x_i \le u_i\}$, which is equivalent to f_i being componentwise concave.

An underestimate of the quadratic constraint can then be constructed by computing facets of the convex envelope for each edge-concave function f_i [63, 64] and a term-wise underestimate for r.

Since an edge-concave function $f : [\ell, u] \to \mathbb{R}$ admits a *vertex polyhedral* [83] convex envelope, it is possible to obtain the value of the convex envelope for $x^\star \in [\ell, u]$ by solving the following linear program [62, 30]:

$$
\begin{aligned}
\min \quad & \sum_i \lambda_i f(v^i) \\
\text{s.t.} \quad & \sum_i \lambda_i v^i = x^\star \\
& \sum_i \lambda_i = 1 \\
& \lambda_i \ge 0 \quad \forall i \in \{1, \dots, 2^n\}
\end{aligned}
$$

where $\{v^i\}$ are all vertices of the hypercube $[\ell, u]$. The corresponding dual solution (α, α_0) leads to an inequality $\alpha^\top x + \alpha_0 \le f(x)$ valid for all $x \in [\ell, u]$. Finally, a valid cut for the original quadratic constraint can be obtained by computing and adding up all these inequalities for the different f_i and underestimating the remaining part r. The cut will be added to the problem if it separates the current point $x^\star$.

Generalized upgrade from quadratic to SOC constraints. A second order cone (SOC) constraint for variables $(x, t) \in \mathbb{R}^{n+1}$ is defined as $||x||_2 \le t$. In some cases, a quadratic constraint $x^\top Qx + d^\top x + b \le 0$, with a symmetric matrix $Q \in \mathbb{R}^{n \times n}$ having a single strictly negative eigenvalue and all others strictly positive, can be written as a second order cone constraint: Using an eigenvalue decomposition, every quadratic constraint can be written as $x^\top PDP^\top x + d^\top x + b \leq 0$, where D is a diagonal matrix containing the eigenvalues and P is orthonormal ($P^{-1} = P^\top$). Say D_{11} is the negative eigenvalue of Q. After performing the change of variables $y = P^T x$, we obtain $y^T Dy + d^T Py + b \leq 0$. This is equivalent to $\sum_{i=2}^n (D_{ii} y_i^2 + \bar{d}_i y_i) + b \leq -D_{11} y_1^2 - \bar{d}_1 y_1$, where $\bar{d} = P^T d$. Completing the squares yields

$$\sum_{i=2}^{n} \left(\left(\sqrt{D_{ii}}\, y_i + \frac{\bar{d}_i}{2\sqrt{D_{ii}}} \right)^2 - \frac{\bar{d}_i^2}{4D_{ii}} \right) + b \leq \left(\sqrt{-D_{11}}\, y_1 - \frac{\bar{d}_1}{2\sqrt{-D_{11}}} \right)^2 + \frac{\bar{d}_1^2}{4D_{11}}.$$

Therefore, we obtain the following extended formulation for the quadratic constraint

$$y = P^\top x, \tag{1}$$

$$\sum_{i=2}^{n} \left(\sqrt{D_{ii}}\, y_i + \gamma_i \right)^2 + c \leq \left(\sqrt{-D_{11}}\, y_1 + \gamma_1 \right)^2. \tag{2}$$

where

$$\gamma_1 = -\frac{\bar{d}_1}{2\sqrt{-D_{11}}}, \quad \gamma_i = \frac{\bar{d}_i}{2\sqrt{D_{ii}}}, \text{ for } i > 1, \quad c = b + \gamma_1^2 - \sum_{i=2}^{n} \gamma_i^2.$$

If $c \geq 0$ and $\sqrt{-D_{11}}\, y_1 + \gamma_1$ has the same sign for all feasible values of y_1, then we can replace (upgrade) the quadratic constraint by linear constraints (1) plus a SOC constraint arising from the square root of (2).

When performing such an upgrade, one has to take care of numerical difficulties. Indeed, since SCIP looks for feasibility with respect to some tolerance, a point may be within the feasibility tolerance for (1) and (2), but fail to be within the feasibility tolerance for the original quadratic constraint. This difficulty arises due to the fact that only after solving the (upgraded) instance, SCIP checks whether the solution satisfies all the original constraints.

To illustrate, consider the constraint $x_1^2 + x_2^2 \leq t^2$ with $t \geq 0$ and let ϵ be the feasibility tolerance. After upgrading, $||x||_2 \leq t$ is obtained and SCIP is going to find a point $(\bar{x}, \bar{t})$ such that $||\bar{x}||_2 \leq \bar{t} + \delta$ with $\delta \leq \epsilon$. This implies that $\bar{x}_1^2 + \bar{x}_2^2 \leq \bar{t}^2 + 2\delta\bar{t} + \delta^2$. This solution is not feasible in the original problem if $2\delta\bar{t} + \delta^2 > \epsilon$.

The way SCIP handles this issue is by scaling the upgraded constraint. Notice that scaling a constraint by L is the same as asking the constraint to be satisfied with tolerance ϵ/L. We have observed that a scaling factor $L = 10$ suffices to resolve all numerical issues in our experiments.

Support for user-defined operators in expression framework (α-state). The framework for handling algebraic expressions has been extended by the introduction of the new operator type `SCIP_EXPR_USER`. This type can be used to implement functions that are not (or not well) supported by the current expression framework.

A "user-function" is essentially defined by several callbacks, some of them are mandatory (see `SCIPexprCreateUser()`). Currently, the following callbacks can or have to be provided: computing the value, gradient, and Hessian of a user-function, given values (numbers or intervals) for all arguments of the function; indicating convexity/concavity based on bounds and convexity information of the arguments; tighten bounds on the arguments of the function when bounds on the function itself are given; estimating the function from below or above by a linear function in the arguments; copy and free the function data. Currently, this feature is meant for low-dimensional (few arguments) functions that are fast to evaluate and for which bound propagation and/or convexification methods are available that provide a performance improvement over the existing expression framework.

The evaluation of Hessians (if provided) is currently only available in dense form and is used by the NLP solver Ipopt. If Hessians are not provided, an approximation algorithm will be enabled in Ipopt. When using user-functions within a nonlinear constraint (`cons_nonlinear`), the callbacks for propagation, convexity/concavity information, and linear under-/overestimation are used for tightening bounds and the linear relaxation. The estimation callback is only used if the user-function is neither convex nor concave.

Note that the current preliminary implementation has not been thoroughly tested yet. Further, larger API changes to improve the user's experience with user-functions, e.g., by bringing the look-and-feel closer to the usual plugin design of SCIP, may occur in a future release of SCIP.

The "user-function" facility can be used with the AMPL interface to mark constraint-defining functions as convex or concave. This is achieved by setting a constraint suffix `curvature` to 1 for convex functions or to 2 for concave functions. The recognition of this suffix needs to be enabled in the source code of `reader_nl.c` first (enable the define for `CHECKCURVSUFFIX`). If a function is marked as convex or concave, it will completely be handled as a user-function, thus its algebraic form is not made available to SCIP. For an application of this (experimental) feature, we refer to [20].

Improved separation procedure for convex quadratic constraints. The separation of convex quadratic constraints $q(x) := x^\top Qx + d^\top x + b \leq 0$, is essential for solving convex MIQCPs by LP-based branch-and-bound. Given a point $\bar{x}$ such that $q(\bar{x}) > 0$, the inequality $q(\bar{x}) + \nabla q(\bar{x})(x - \bar{x}) \leq 0$ separates $\bar{x}$ from the feasible region. This type of cut is called a *gradient cut* at $\bar{x}$. However, gradient cuts at infeasible points are not necessarily tight at the feasible region, meaning that the hyperplane $q(\bar{x}) + \nabla q(\bar{x})(x - \bar{x}) = 0$ is not supporting the region $S = \{x \colon q(x) \leq 0\}$ when $\bar{x} \notin S$.

If an interior point x_0 of S is known, a binary search between $\bar{x}$ and x_0 can be performed until a point in the boundary of S is reached. A gradient cut at this point is tangent at the feasible region and separates the point $\bar{x}$. Specifically, the equation $q(\lambda\bar{x} + (1-\lambda)x_0) = 0$ for $\lambda \in [0, 1]$ needs to be solved. With the new parameter `constraints/quadratic/gaugecuts`, which is enabled by default, the user can tell SCIP to generate the cuts previously described.

Improvements of optimization-based bound tightening. Optimization-based bound tightening (OBBT) is one of the most effective procedures to reduce variable domains of nonconvex mixed-integer nonlinear programs (MINLPs) [70]. The strategy of OBBT is to minimize and maximize each variable over a linear relaxation of the problem (OBBT-LP), which provides the best possible bounds that can be learned from the linear relaxation. In general, these are not the best bounds for the nonconvex MINLP. By using the new parameters `propagating/obbt/separatesol` and `propagating/obbt/propagatefreq` it is possible to trigger additional separation and propagation rounds after solving an OBBT-LP. Typically, this increases the quality and quantity of bound reductions found by the OBBT propagator and thus improves the linear relaxation of the problem.

2.6 SOS1 constraint handler

A *special ordered set (SOS)* constraint of type 1 requires that at most one out of a given set of variables takes a nonzero value. SCIP can handle SOS1 constraints without the use of a mixed-integer programming formulation involving additional binary variables. The SOS1 constraint handler represents every constraint as a clique of a conflict graph. This structure can be algorithmically exploited, e.g., to get improved branching rules, cutting planes, and primal heuristics. The most significant changes of the current release are briefly explained below but for all details we refer the reader to [26].

Branching rules. The validity of the SOS1 constraints can be enforced by different branching rules. In case that the SOS1 constraints are non-overlapping, i.e., do not share variables, the algorithm automatically applies the classical *SOS1 branching* rule [8], which was already available prior to version 3.2. In the overlapping case, SOS1 branching can be improved by considering complete bipartite subgraphs (not necessarily induced) of the conflict graph:

With standard settings the algorithm branches on the neighborhood of a single variable x_i, i.e., in one branch x_i is fixed to 0 and in the other all its neighbors $\Gamma(i)$ in the conflict graph. This means that the considered complete bipartite subgraph is associated with a node partition $C_1 \dot\cup C_2$ with $C_1 = \{i\}$ and $C_2 = \Gamma(i)$. The new bipartite branching method of the constraint handler (turned on by setting the parameter `constraints/sos1/bipbranch` to `TRUE`) searches for more general complete bipartite subgraphs with $|C_1| > 1$ and $|C_2| > 1$. A valid decomposition of the solution set is derived by adding domain fixings $x_j = 0$ for every $j \in C_1$ on one branch and $x_j = 0$ for every $j \in C_2$ on the other. Bipartite branching can be beneficial if the conflict graph contains large balanced complete bipartite subgraphs.

Furthermore, with the new option `constraints/sos1/addcomps`, the algorithm tries to strengthen the formulation of the branching nodes by finding local *complementarity constraints* of the form $x_i \cdot x_j = 0$. Adding local complementarity constraints results in a nonstatic conflict graph, which may change dynamically with every branching node.

Cutting planes. The recent SCIP release includes two new cutting plane separators for SOS1 constraints as well as improvements to the separator of clique bound inequalities, which was already available prior to version 3.2.

- *Clique bound inequalities:* Consider an SOS1 constraint defined over a variable (sub-)set S on variables x_i, $i \in S$. If every variable has a finite upper bound $x_i \leq u_i$ with $u_i \in \mathbb{R}_{>0}$ for all $i \in S$, one can add a bound inequality of the form $\sum_{i\in S} \frac{x_i}{u_i} \leq 1$ to the LP relaxation. Instead of separating bound inequalities from the initial SOS1 constraints, a new feature of the recent release allows to separate them by searching heuristically for maximum weighted cliques in the conflict graph. This approach can significantly improve the performance of the solver in case that the SOS1 constraints overlap. If the default settings are used, bound inequalities are separated with node depth frequency of 10, but a user-defined frequency can be set with the parameter `constraints/sos1/boundcutsdepth`.

- *Implied bound inequalities:* Savelsbergh [75] describes preprocessing and probing techniques to deduce logical implications between variables. These techniques can be adapted to problems containing SOS1 constraints by detecting implications of the form

$$z > 0 \Rightarrow x \leq \lambda,$$

  where x and z are variables and $\lambda \in \mathbb{R}$. If $x \leq u$ and $z \leq d$ with u, $d \in \mathbb{R}_{>0}$, then one can deduce an implied bound inequality $x + \frac{u-\lambda}{d}\, z \leq u$. The inequalities are generated as part of the separation routine of the SOS1 constraint handler in SCIP with a node depth frequency indicated by the parameter `constraints/sos1/implcutsdepth`.

- *Disjunctive cuts:* SCIP additionally incorporates a new separator `sepa_disjunctive`. It separates disjunctive cuts for two-term disjunctions of the form $x_i \leq 0 \vee x_j \leq 0$, which originate from edges $\{i, j\}$ of the conflict graph. The implementation in SCIP is based on the approach presented in [48], where the authors describe a cost-effective way to generate disjunctive cuts directly from the simplex tableau.

Diving heuristics. Standard MIP-based diving rules iteratively tighten a bound of a fractional variable and resolve the LP (see Section 2.3). However, these rules are not directly applicable to SOS1 constraints whose variables can be continuous. In the current SCIP release some of the diving heuristics have been made compatible with SOS1 constraints by imitating diving on the binary variables y_i of a MIP-reformulation with big-M constraints $x_i \leq u_i\, y_i$, where $u_i \in R_{>0}$. The candidate that is selected for diving depends on the specific diving heuristic; e.g., in case of fractional diving, the variables are selected with respect to the ratio x_i^*/u_i, where x_i^* is the current LP value of x_i.

2.7 Further changes

Extended probing mode. SCIP contains two methods to do a temporary dive in the branch-and-bound tree. The *diving mode* works directly on the LP structure and allows to solve the LP after possibly changing variable bounds, adding rows, or changing the objective function. In contrast, the *probing mode* creates temporary branch-and-bound nodes, which makes it possible to perform backtracks during a dive. It allows to perform domain propagation at the temporary nodes and provides the ability to solve the LP by column generation if a pricer plugin is implemented. The probing functionality has been extended in two ways: First, cutting planes can be separated during probing using `SCIPseparateSol()` and then applied to the current LP using `SCIPapplyCutsProbing()`. Second, objective function coefficients can now be changed during probing via `SCIPchgVarObjProbing()`. Those changes are mapped to the temporary nodes, so that a backtrack will automatically undo objective changes as well. Note that due to global conflicts potentially created by conflict analysis, domain propagation is always performed with respect to the original objective function.

Transfer and merging history information. Several primal heuristics of SCIP create and solve sub-MIPs that result from fixing variables and/or adding constraints. Prior to this release, variable information was completely uninitialized inside these sub-MIPs no matter how much history information had already been collected during the main search. A transfer of collected variable information can be beneficial in both directions; the created sub-MIPs are solved with a reasonable initialization of history information, while the main SCIP might benefit from additional history information collected during the sub-MIPs. We call information passing to a sub-MIP instance a *transfer*, while it is called a *merge* in the opposite direction. The merge of history information is limited to sub-MIPs that use the same objective function for the variables and do not add additional constraints except an objective cutoff. Transfers and merges are currently controlled centrally by the user parameters `history/allowtransfer` and `history/allowmerge` that determine whether sub-MIP copies of the problem should be initialized with the main history information and if the information should be merged back from sub-MIPs, respectively. The latter is only performed in plugins that define sub-problems solely through variable fixings, but no additional constraints or changed objective function.

Using sparsity information of the SoPlex LP. The use of sparsity information within separators has been implemented as a performance improvement for this release. Many separators use one-row relaxations of the problem which are obtained as aggregations of the rows of the constraint matrix. Normally, the aggregation coefficients are the solution of a linear system with the current basis matrix that is computed by the LP solver If SoPlex is used as LP solver, sparsity information of the array of aggregation coefficients is directly available and exploiting it leads to a significant reduction of the execution time for various separators. The implementations of complemented mixed-integer rounding cuts [59], flow cover cuts [58], Gomory mixed-integer rounding cuts [41, 40], multi-commodity flow network cuts [6], strong Chvátal-Gomory cuts [56], $\{0, 1/2\}$-Chvatal-Gomory cuts [18], and Chvátal-Gomory cuts computed via a sub-MIP [15, 28] were affected by the improved use of sparsity information.

Handling different definitions of infinity in SCIP and SoPlex. SCIP and SoPlex use different default values for infinity, namely 10^{20} and 10^{100}, respectively. Transferring floating point values between the two codes is now better handled by properly translating them into the respective environment. For instance, a bound that is set to 10^{20} in SCIP will be set to 10^{100} when copying the problem to SoPlex.

Memory management. The buffer memory system was unified to be compatible with the standard/block memory of SCIP. Moreover, safety checks for integer overflows in the size of allocated buffer memory were added. Additionally, a new type of buffer memory was added, called clear buffer. It can be used if a buffer array is needed with all entries initialized to zero. Before freeing this buffer array, all non-zero entries need to be reset to zero, which can often be performed more efficiently than by overwriting the complete array, for example if only few entries were modified and their indices are known.

Furthermore, the Zi round heuristic has been modified to now use buffer arrays to store its required data instead of permanent block memory arrays. This change significantly reduces the overall memory footprint of SCIP.

Python interface. In SCIP 3.2.1, the already existing interface to the programming language Python has been vastly extended. The interface is written in Cython [91] and enables the user to model mixed-integer linear and quadratic programming problems. It also supports SOS1 and SOS2 constraints. Constraints can be conveniently created by means of expressions:

```
from pyscipopt import Model
scip = Model()
x = scip.addVar('x', vtype='CONTINUOUS')
y = scip.addVar('y', vtype='INTEGER')
scip.setObjective(x + 3y)
scip.addCons(2x - y*y >= 10)
scip.optimize()
```

Furthermore, new SCIP plugins can be written in Python. To this end, base classes with empty callback methods are provided and a derived class can implement the relevant functionality in Python by overriding the callbacks. This feature enables the user to do fast prototyping of new algorithmic ideas without the coding overhead of the C language. Among the supported plugins are `pricers`, `heuristics`, `presolvers`, `separators`, and `constraint handlers`.

The code uses *docstrings* [92], so calling `help()` on a method will show its documentation. The most important interface methods are implemented, but there are methods and data structures in SCIP that currently cannot be called or used from Python.

AMPL interface. The variable and constraint attributes (flags in `SCIPcreateVar()` and `SCIPcreateCons()`) of SCIP can now be set via AMPL suffixes, where 0 (unset) denotes the default for that attribute, 1 denotes `TRUE`, and other values denote `FALSE`. For variables, the flags "initial" and "removable" are recognized. For constraints, the AMPL interface recognizes the flags "initial", "separate", "enforce", "check", "propagate", "dynamic", and "removable".

SCIP output. *Branch-and-bound Analysis Kit:* SCIP can output information on the branch-and-bound tree to a file that can be fed into the Branch-and-bound Analysis Kit (BAK), see [66]. This option is activated by supplying a filename with the SCIP parameter `visual/bakfilename`.

Feasibility Check: For the feasibility check of the best found solution it is possible to use the parameter `display/allviols` in order to display all violated constraints and variable bounds. SCIP also provides the possibility to increase the feasibility tolerance for this check by setting the factor `numerics/checkfeastolfac`. This can be very useful for numerically difficult problems for which one wants to use a given feasibility tolerance during the solving process but a larger one for the final feasibility check.

Problem Permutation: SCIP release 2.0 introduced the possibility to randomly permute the order of variables and constraints during problem transformation. Release 3.2 now allows the permutation of the original problem directly after reading. This is triggered if the parameter `misc/permutationseed` is set to a non-negative value before reading in the problem. An instance modified in this way often leads to different solving behavior, a phenomenon called *performance variability* [52]. In order to get more reliable benchmarking results, the MIPLIB 2010 test scripts [90] use the new SCIP functionality to automatically permute the input problems with SCIP, write them to an LP file, and then perform computational experiments on these permuted instances.

3 SoPlex: Sequential object-oriented simPlex

This release of the SCIP Optimization Suite contains version 2.2.0 of the LP solver SoPlex. Beside performance improvements of the simplex implementation, the new version provides the possibility to compute arithmetically exact solutions of LPs with rational input data.

Exact solving capability. Algorithmically, the exact solving capability relies on two methods that are interleaved with SoPlex's iterative refinement scheme:

- a rational LU factorization of the basis matrix that is used to compute the exact rational solution corresponding to the current basis. This provably verifies the feasibility and optimality of the current candidate basis;
- a rational reconstruction procedure based on continued fraction approximations which "rounds" the current numeric solution to a nearby solution with low denominators.

These methods can be used individually or in combination to solve LPs exactly within a theoretical worst-case bound of polynomially many refinement rounds. For rational reconstruction, we have integrated and adapted code of Daniel Steffy.[1] For more details, including a computational study of the performance of both approaches, see [37]. For more details on the underlying iterative refinement method see [38] and [39].

By default SoPlex still operates as a floating-point LP solver. The exact solving features are activated by parameters. See the section "How to use SoPlex as an exact LP solver" of SoPlex's online documentation under http://soplex.zib.de for more details.

Sparsity exploitation. Sparse data structures are now used in more parts of the code to avoid unnecessary operations on zero-valued entries. Keeping track of nonzero values in a vector also speeds up its clearing procedure, since only a fraction of the indices needs to be touched. The same principle applies to the computation of the current objective value, which is required to determine whether an objective limit has been reached. Here, the nonbasic part of the objective can be initialized once and then be updated with respect to the simplex pivot in the current iteration.

In the pricing step, *hyper sparse pricing* (see [50, 42]) is enabled by default: Better performance is achieved using this pricing by restricting the evaluated indices to a small set of interesting candidates. This set is updated whenever new candidates are available due to a basis change.

Further developments. The bound flipping ratio test for the dual simplex is enabled by default and its potential is increased by preserving bounds of boxed variables in presolving. Steepest edge pricing norms can be set and extracted, e.g., for restoring them after a dive in the MIP branch-and-bound tree. To ensure thread safety, the message handler is no longer a global variable.

4 ZIMPL: Zuse Institute Mathematical Programming Language

A single change has been made for the 3.3.3 release of ZIMPL. For easy compilation under Windows, an NMAKE-Makefile was added to the distribution.

[1]Daniel E. Steffy. Dense Iterative Refinement Solver Version 1.1, https://files.oakland.edu/users/steffy/web/rational/, accessed January 2015

5 UG: Ubiquity Generator framework

The parallelization framework UG has been developed and tested extensively with SCIP as the base solver. In the new version, UG 0.8, the dynamic load balancing mechanism has been improved, especially for the ramp-down process. With the configuration ug[SCIP,MPI], which is ParaSCIP, a successful test using 80,000 cores was run on the supercomputer Titan at the Oak Ridge National Laboratory[2]. This run was instrumental in solving the open MIPLIB2010 instance `rmine10` [52]. See [81] for details about the computational results and the algorithmic improvements.

A major new feature is the functionality to parallelize a customized SCIP solver. To realize this feature ug[SCIP, Pthreads] (shared memory parallelization) and ug[SCIP, MPI] (distributed memory parallelization) are provided as libraries. This release of the SCIP Optimization Suite includes an example demonstrating the use of these libraries to parallelize a customized SCIP solver for solving Steiner Tree Problems, SCIP-Jack [32], see also Section 7.

6 GCG: Generic Column Generation

This release of the SCIP Optimization Suite contains version 2.1.0 of the generic branch-and-price code GCG.

Basis cuts separator. In branch-price-and-cut algorithms, cutting planes formulated with original variables do not change the structure of the pricing problems, whereas cutting planes formulated with master variables in general do. Since projecting a basic solution of the master LP relaxation to the original solution space does not necessarily yield a basic solution, cutting planes in the original problem that are separated using a basis are not directly applicable. Range [73] (and others) proposed as a remedy to heuristically compute a basic solution and separate this auxiliary solution also with cutting planes that stem from a basis. This might not only cut off the auxiliary solution, but also the solution we originally wanted to separate. If the solution we wanted to separate is not cut off, we can strengthen the original LP relaxation by temporarily adding the obtained cutting planes to the problem formulation and repeat the procedure.

This separation algorithm was implemented in GCG and added to the latest release. For a detailed description of the separation algorithm (including several heuristics to obtain a basic feasible solution) as well as computational results we refer to [57].

Column pool. A column pool was added to GCG in the current release. At present the column pool is mainly used for column management. Columns are not immediately added to the restricted master problem but are collected in a column pool which is searched for negative reduced cost columns after solving the pricing problems. In the future, we plan to keep promising columns in the column pool if they did not enter the LP and search the pool in later rounds before calling the pricer.

For each column in the column pool, we store the current reduced cost and the age, which is defined as the number of pricing steps since its creation. For the column pool, both a soft and a hard limit on the number of columns in the column pool is specified. The hard limit is always obeyed, whereas the soft limit can be temporarily violated. The column pool is implemented as a priority queue where columns with small reduced cost are prioritized. In the beginning of each pricing step, the reduced costs of the columns in the column pool are updated and the priority queue is reinitialized. After the pricing problems are solved, columns are added to the column pool. Then, variables having negative reduced cost are added to the restricted master problem and removed from the column pool. In the end, the oldest columns are removed from the column pool such that the soft limit for the number of columns in the column pool is obeyed.

[2]https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/

To avoid checking for feasibility of columns in each node of the branch-and-bound tree, the column pool will be cleared whenever the node changes.

Set covering heuristic. In several cases, a given MIP model can be reformulated via Dantzig-Wolfe decomposition such that the resulting master problem takes the form

$$\begin{aligned} \min \quad & c^\top \lambda \\ \text{s.t.} \quad & A\lambda \geq 1 \\ & \lambda \in \{0,1\}^n, \end{aligned} \tag{3}$$

with $A \in \{0,1\}^{m\times n}$, $c \in \mathbb{R}^n_+$. A heuristic by Caprara et al. [19] was added to GCG that exploits the structure of such a *set covering problem*.

Based on the Lagrangian relaxation of (3), it iteratively computes a sequence $u^k \in \mathbb{R}^m_+$ of near-optimal Lagrangian multiplier vectors for the Lagrangian subproblem

$$\begin{aligned} \min \quad & c^\top \lambda - u^\top (A\lambda - 1) \\ \text{s.t.} \quad & \lambda \in \{0,1\}^n. \end{aligned} \tag{4}$$

For each multiplier, a heuristic solution x^k for (3) is then computed by greedily setting λ variables to 1, using a score function taking the multiplier into account. The best of these solution is then added to the solution pool of GCG.

Refactoring. The constraint handlers that coordinate the branching trees of the original problem and the master problem have undergone some refactoring: Some fields in their data structures and methods were renamed, and two methods were split into smaller ones. Some documentation was also added to the code.

Further, the memory consumption of the cutpacking detector was reduced by decreasing its excessive usage of the `SCIP_HASHMAP` data structure.

Further developments. The numerical tolerances given to the original problem are now also used in the master problem as well as in the pricing problems. The compiler flag `CPLEXSOLVER` was re-added to GCG. When this flag is set to `true`, the pricing problems that are solved as a MIP are solved with CPLEX instead of SCIP.

7 Other extensions

The SCIP distribution contains example projects that are mainly designed as a starting point for new users. Some show how to use the API for specific types of problems, others are simple extensions of SCIP illustrating the implementation of a branch-and-price or branch-and-cut approach for a specific problem. In recent years, some complex extensions were developed. It was decided for the current release that these more complex examples are shipped with the distribution. However, they do not primarily serve as an example, but rather extend the applicability of SCIP to new problem classes, constituting competitive codes for specific applications. To reflect their function as a practical extension of SCIP a new category for extensions has been introduced. These extensions are collected in the *applications* subdirectory of the distribution.

The existing vertex coloring and scheduler examples were moved to the applications directory and two new applications were added. These consider Steiner tree problems and multi-objective optimization and are introduced in the following section.

Moreover, SCIP-SDP, the external mixed-integer semidefinite programming extension of SCIP was significantly improved, as is discussed at the end of this section.

7.1 SCIP-Jack: Steiner tree problems and variants

The *Steiner tree problem in graphs* (*SPG*) is a classical $\mathcal{NP}$-hard problem [49]: Given an undirected connected graph $G = (V, E)$, weights $c : E \to \mathbb{Q}_+$ and a set $T \subseteq V$ of *terminals*, the problem is to find a minimum weight tree $S \subseteq G$ that spans T. Although extensively studied both theoretically and practically, the classical SPG rarely arises when modeling real-world problems. Instead, one predominantly encounters variations of the classical Steiner tree problem. An example of such a variation is the *prize-collecting Steiner tree problem*: a feasible solution to this problem can span any subset of the terminals, but concomitantly a non-negative penalty is charged for each terminal not contained.

The SCIP 3.2 release sees the first distribution of the Steiner tree problem solver SCIP-Jack [32], which can handle both the classical SPG and ten of its variants. This versatility is rendered possible by transformations of the different Steiner tree problem variants into a general form that are then solved by a branch-and-cut algorithm. In fact, this general form is a *directed Steiner tree problem* (V, A, T, c, r) possibly incorporating additional constraints. As compared to the SPG the directed Steiner tree problem contains arcs A instead of edges and furthermore distinguishes a terminal r that is required to be the root of each feasible Steiner tree. An SPG can be transformed into a directed Steiner tree problem by replacing each edge by two anti-parallel arcs of the same cost and distinguishing an arbitrary terminal as the root.

The solving approach employed within SCIP-Jack is based on the following *flow-balance directed cut* IP formulation, which associates with each arc $a \in A$ a variable y_a, indicating whether a is contained in the Steiner tree ($y_a = 1$) or not ($y_a = 0$):

$$\min \quad c^\top y \tag{5}$$

$$y(\delta^-(W)) \geq 1 \qquad \forall\, W \subset V : r \notin W, W \cap T \neq \emptyset, \tag{6}$$

$$y(\delta^-(v)) \begin{cases} = & 0, \text{if } v = r; \\ = & 1, \text{if } v \in T \setminus \{r\}; \\ \leq & 1, \text{if } v \in N; \end{cases} \qquad \forall\, v \in V, \tag{7}$$

$$y(\delta^-(v)) \leq y(\delta^+(v)) \qquad \forall\, v \in V \setminus T, \tag{8}$$

$$y(\delta^-(v)) \geq y_a \qquad \forall\, a \in \delta^+(v), v \in V \setminus T, \tag{9}$$

$$y_a \in \{0, 1\} \qquad \forall\, a \in A, \tag{10}$$

where $\delta^+(X) := \{(u, v) \in A : u \in X, v \in V \setminus X\}$ and $\delta^-(X) := \delta^+(V \setminus X)$ for $X \subseteq V$, and $y(F) := \sum_{e \in F} y_e$ for $F \subseteq E$. Further details of the formulation are provided in [53]. Since the model potentially contains an exponential number of constraints, a separation approach is employed. Violated constraints are separated during the execution of the branch-and-cut algorithm.

Considering the implementation, the two underlying plugins are a *reader* to read problem instances and possibly transform them, and a *problem data* to store the graph and build the model within SCIP. With the problem having been read in and transformed, the solving approach can be dissected into three main components:

The heart of SCIP-Jack is a *constraint handler* that checks solutions for feasibility and separates violated model constraints. SCIP provides a filtering of cuts to improve numerical stability and dynamic aging of the generated cuts. Additionally, general-purpose separation methods of SCIP such as Gomory and mixed-integer rounding cuts are employed. Concerning branching, the customary hybrid branching rule [3] of SCIP is used; node selection is performed with respect to a best estimate criterion—interleaved with best bound and depth-first search phases [1].

The second pillar of SCIP-Jack is constituted by reduction methods for both the SPG and, to a lesser extent, for its variations. These methods are performed prior to the model being built within SCIP. Additionally, a Steiner tree problem specific propagator has been implemented based on an idea first published in [25].

As the third major solving component, three Steiner tree problem specific primal heuristics have been implemented in SCIP-Jack as separate plugins: First a constructive heuristic based on [24] and [68], second a local search approach described in [84], and third a new recombination heuristic [32].

SCIP-Jack was able to take first place at the 11th DIMACS Challenge in the category *rooted prize-collecting Steiner tree problem.* Moreover, employing the UG framework we were able to solve three SPG benchmark instances for the first time to optimality and improve the best known solutions to another 11 instances [34].

Finally, to the best of our knowledge, this release marks the first time that a powerful exact Steiner tree solver has been available in source code to the scientific community. Furthermore, we plan to publish a considerably extended version of SCIP-Jack as part of the next SCIP bugfix release. By virtue of a variety of new presolving routines (see [74]) and furthermore a dual heuristic to decide on a good initial LP relaxation, the performance of SCIP-Jack on almost all Steiner tree variants will be drastically improved. In this way, SCIP-Jack will be able to outperform other (specialized) state-of-the-art solvers on several problem variants.

7.2 PolySCIP: Multi-criteria MIP and LP

PolySCIP [95] is a solver for multi-criteria integer programs as well as multi-criteria linear programs, see Fig. 2 and Fig. 3 for an illustration, respectively. In other words, it solves optimization problems of the form:

$$\begin{aligned} \min\ & (c_1^\top x, \dots, c_k^\top x) \\ \text{s.t. } & Ax \le b, \\ & x \in \mathbb{Z}^n \vee \mathbb{Q}^n, \end{aligned}$$

where $k \ge 2,\ A \in \mathbb{Q}^{m\times n}$, $b \in \mathbb{Q}^m$.

Let $\mathcal{X}$ be the feasible space of the considered problem and $\mathcal{Y} = \{(c_1^\top x, \dots, c_k^\top x) : x \in \mathcal{X}\}$ be the corresponding image in objective space. A point $y^* \in \mathcal{Y}$ is an extreme supported non-dominated (ESN) point if there is a positive weight vector $w \in \mathbb{R}^k$ such that $w^\top y^* < w^\top y$ for all $y \in \mathcal{Y} \setminus \{y^*\}$ (in the case of minimization; $w^\top y^* > w^\top y$ in the case of maximization). The algorithmic approach and methodology of PolySCIP to compute all ESN points can be summarized as follows (for a minimization problem): At the beginning an arbitrary ESN point $y_1 \in \mathcal{Y}$ is computed (by lexicographic optimization) or it is determined that there do not exist any ESN points for the considered problem. Let $\bar{\mathcal{Y}}$ be the set of ESN points computed (after the first iteration) and let $P = \{(a, w) \in \mathbb{R} \times W\ :\ a \le w^\top y\ \forall y \in \bar{\mathcal{Y}}\}$ be the partial weight space polyhedron with $W = \{w \in \mathbb{R}^k_+ : \sum_{i=0}^k w_i = 1\}$ (At this stage consider all vertices of P as non-marked). In every next iteration a non-marked vertex $(a, w) \in P$ is chosen and the weighted optimization problem $\min w_1 c_1^\top x + \ldots + w_k c_k^\top x$ s.t. $x \in \mathcal{X}$ is solved. Let $\tilde{x}$ be the computed solution and let $\tilde{y}$ the corresponding point in objective space. If $a > w^\top \tilde{y}$, then $\tilde{y}$ is a new ESN point and $\bar{\mathcal{Y}}$ and the partial weight space polyhedron P is altered. If $a \le w^\top \tilde{y}$, then the vertex $(a, w) \in P$ is marked. The algorithm stops if all vertices of P are marked.

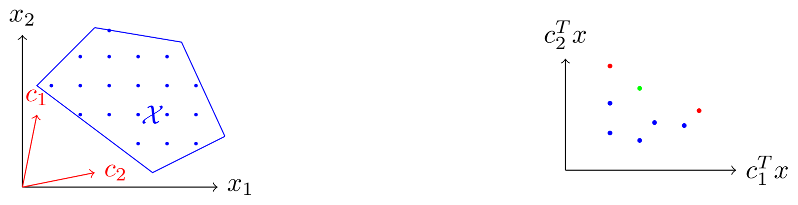

Figure 2: Feasible space of bi-criteria integer program and set in objective space with dominated points (blue), supported non-dominated points (red) and non-supported non-dominated point (green).

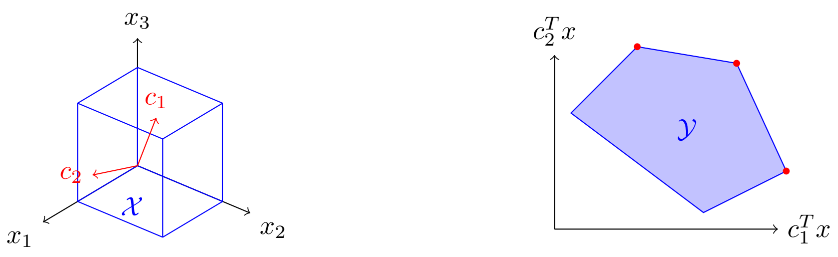

Figure 3: Feasible space of bi-criteria linear program and set in objective space with extreme non-dominated points (red).

The current version of PolySCIP computes all ESN points (and non-dominated rays) given a multi-criteria linear program or multi-criteria integer program, respectively. For multi-criteria mixed-integer programs, computing the non-dominated points is more complicated and is currently not fully supported by PolySCIP.

The file format (with suffix .mop) is based on the MPS file format. Objectives are considered to be non-constrained rows, i.e, they are defined by the key 'N' in the 'Rows' section. For instance, (the beginning of) a tri-criteria minimization problem named 'sustain' with objectives *COSTS*, *EMISSION* and *HAZARDS* would start with

```
NAME     sustain
OBJSENSE
 MIN
ROWS
 N  COSTS
 N  EMISSION
 N  HAZARDS
```

followed by the usual inputs for constraints, upper and lower bounds et cetera defined by the original MPS format.

7.3 SCIP-SDP: A mixed-integer semidefinite programming solver

Together with SCIP 3.2, version 2.0 of SCIP-SDP [97] was released. SCIP-SDP is a plugin for SCIP to solve mixed-integer semidefinite programs (MISDPs) of the form

$$\begin{aligned} \min \quad & b^\top y \\ \text{s.t.} \quad & \sum_{i=1}^{m} A_i y_i - C \succeq 0, \\ & y_i \in \mathbb{Z} \quad \forall i \in I, \end{aligned} \tag{11}$$

for symmetric matrices C, $A_i \in \mathbb{R}^{n \times n}$ for all $i = 1, \ldots, m$ and a set of indices of integer variables I. SCIP-SDP adds a constraint-handler for semidefinite constraints to SCIP as well as a relaxator that can solve semidefinite programs via an interface to SDP-solvers. It also includes a file-reader for an enhanced sparse SDPA-format with added lines for integrality constraints.

The original version 1.0 was developed by Sonja Mars and Lars Schewe [60, 61]. The current version involves a redesign of the code, including some porting from C++ to C, and several new components, as described in the following.

For SCIP-SDP 2.0 a two-level SDP-solver interface was added, which is, in principle, independent of SCIP. The first level general interface is shared among the different SDP-solvers and takes care of some of the presolving, which is usually not contained in SDP-solvers. This includes the removal of locally fixed variables and zero rows and columns in the semidefinite constraint in (11), which, for instance, might occur by fixing all variables with entries in these rows and columns to zero. The second level interface is specific to the different SDP-solvers. It passes the locally presolved problems to the solver, sets parameters accordingly and then calls the SDP-solver. In addition to the interface to DSDP [10], which was already included in version 1.0, an interface to the SDP-solver SDPA [87, 88] was added. When using the latter interface, the different standard settings provided by SDPA from "fastest" to "stable" are iteratively applied. If none of these settings is able to solve the problem, a penalty formulation like in DSDP is used, where an identity matrix scaled by an additional slack variable, which is penalized in the objective, is added to the semidefinite constraint in (11).

Additionally, a dual fixing propagator called `sdpredcost` was added to SCIP-SDP. This is a generalization of reduced cost fixing for linear programs that tightens variable bounds and fixes binary variables based on the values of the dual variables corresponding to variable bounds in the solutions of the semidefinite relaxations. For a variable y_j with upper and lower bounds u_j and ℓ_j, dual variables v_j and w_j corresponding to these variable bounds, the upper bound U on the minimization problem given by the best known integer solution, and the optimal objective value $\bar{f}$ of the semidefinite relaxation, one can show that the following holds for all optimal solutions y:

$$y_j \geq u_j - \frac{U - \bar{f}}{v_j} \quad \text{and} \quad y_j \leq \ell_j + \frac{U - \bar{f}}{w_j}.$$

This propagator can be restricted to either only binary or integer and continuous variables by changing the parameters `forbins` and `forintcons`. With standard settings the propagator is applied to all variables.

Finally SCIP-SDP now uses the SCIP shell, cutoff-bounds can be transferred to the SDP-solvers by enabling the parameter `relaxing/SDP/objlimit` and average and total SDP iterations are now displayed in the shell instead of LP iterations. Furthermore, the output of the SDP-solver can be enabled by the parameter `relaxing/SDP/sdpinfo`.

8 Summary of performance improvements

This paper highlights the most important features and improvements of the latest release of the SCIP Optimization Suite. While some features extend the range of problems that can be solved with the SCIP Optimization Suite, other features are targeted towards improving performance on particular instance classes. A particular focus is the performance on the problem classes most frequently solved by the SCIP Optimization Suite: mixed-integer linear programs (MIPs) and mixed-integer nonlinear programs (MINLPs).

Figure 4 compares the performance of SCIP 3.2 against previous major SCIP versions on the MIPLIB 2010 [52] benchmark set. For each SCIP version presented in Figure 4, the most recent release of SoPlex at the time was used as the LP solver. As can be seen, the number instances solved within the time limit of two hours increases by three and the average running time decreases by 24 %. Note that this test set of 87 instances can only provide a rough picture for the performance of a MIP solver. The performance improvement achieved by version 3.2 on single instances or problems of a specific class may differ. Nevertheless, the MIPLIB benchmark set was specifically designed to cover a variety of different problem types and is widely used for benchmarking purposes [65].

For MINLPs, we observe a performance improvement of 14 % and a node reduction of 31 % on the MINLPLib2 [89] benchmark set. A standard performance profile between SCIP 3.2 and the previous version 3.1 on all 1357 instances of the MINLPLib2 is presented in Figure 5. As a result of the performance improvements, SCIP 3.2 now solves 31 more instances than SCIP 3.1. However, SCIP 3.1 achieves a faster runtime on more instances than SCIP 3.2. This can be explained by the addition of new features aimed at solving hard instances, which may create a performance overhead on easy instances. Overall, for hard instances this overhead pays off and results in more instances being solved.

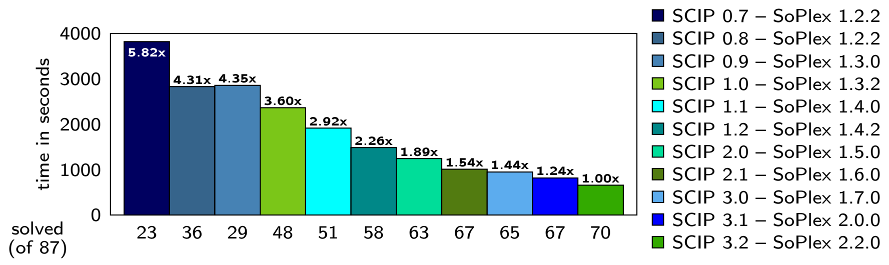

Figure 4: Version to version performance improvement on MIPLIB 2010 [52], relative to the latest release. Shown timings are shifted geometric means over all 87 instances. On the horizontal axis the numbers of solved instances are displayed.

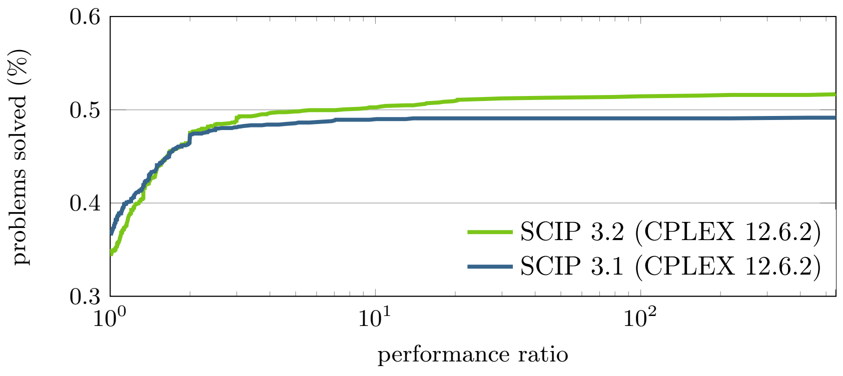

Figure 5: Performance profile for SCIP 3.1 and 3.2 over 1357 instances of the MINLPLib2 [89].

9 Outlook

The components of the SCIP Optimization Suite are actively developed in many directions. Within SCIP, special focus lies on the extension of the reoptimization capabilities, on improved branching rules, and MINLP. The reoptimization feature of SCIP will be extended to the case of general MIP. Additionally, the main research topics are new heuristics to reduce the size of the stored search tree, better concepts to deal with dual reductions in branching, and improved compatibility with some advanced MIP techniques like conflict analysis.

In the context of branching rules, further work on cloud branching [13] is planned, as well as the integration of other recent branching developments like nonchimerical branching [29] and an abstract model for the scoring function used in branching [54]. A dynamic solving strategy for SCIP based on the concept of solving phases [46] will be made available.

An important class of MINLPs is the class of mixed-integer quadratically constrained programs (MIQCPs). Future goals for the improvement of SCIP on MIQCPs include the implementation of techniques from the literature that are not yet available in SCIP. Examples are the reformulation-linearization technique of Sherali and Adams [78] and valid inequalities arising from semidefinite programming reformulations [79, 76, 77]. Furthermore, the next major release of SCIP will be able to use a convex nonlinear relaxation to obtain stronger dual bounds at nodes of the branch-and-bound tree.

In SoPlex, besides general performance improvements, a solution polishing feature is in development to promote integrality for solutions of MIP relaxations.

Complementary to the external parallelization scheme available via the UG framework, infrastructure for an internal parallelization of SCIP is currently being developed. This will both enable the parallelization inside plugins and allow the parallel call of similar plugins. Within the UG framework, the focus is on the parallelization of additional extensions of SCIP and the interaction with the future internal parallelism of SCIP.

In the near future, the main research direction in GCG will be the further development of structure detection. First, it will be modularized in the sense that a structure detector or the user will be able to only specify a partial structure which can then be completed by the (other) detectors. Second, the effect of different structures on the solution process will be further investigated, in the hope to devise a better a priori estimate which structure is the best suitable for a given problem.

The development of the extensions presented in Section 7 will be continued. The main research directions of PolySCIP are the computation of non-supported non-dominated points for problems with an arbitrary number of objectives and the computation of the front of non-dominated points of multi-criteria mixed-integer problems. For SCIP-SDP, further research directions include linear and conic cuts as well as warmstarting possibilities. For the next release, SCIP-SDP will be extended by a fractional diving heuristic for MISDPs. Furthermore, an interface to the commercial SDP solver of MOSEK [94] is currently in development.

Acknowledgements. The authors want to thank all contributors to the SCIP Optimization Suite. Special thanks go to Roland Wunderling, the creator of SoPlex, Tobias Achterberg, the creator of SCIP, the former developers of SCIP, Timo Berthold, Stefan Heinz, and Kati Wolter, and the former GCG developer Martin Bergner. We are grateful for the support of Alexander Martin who developed SIP, the predecessor of SCIP, and who fostered many of the latest developments in presolving and to Marco Lübbecke for his vision of a generic branch-and-price solver and his invaluable encouragement and advice concerning the development of GCG.

The work for this article has been partly conducted within the *Research Campus Modal* funded by the German Federal Ministry of Education and Research (fund no. 05M14ZAM). The third and tenth author would like to thank the German Research Foundation (DFG) for funding since part of this research has been carried out within the Collaborative Research Center 805. The development of PolySCIP started in the project A5 *Multicriteria Optimization* within the Collaborative Research Center 1026 *Sustainable Manufacturing*, which the thirteenth author would like to acknowledge.

We are grateful to the HLRN III supercomputer staff, especially Matthias Läuter and Guido Laubender and to the ISM supercomputer staff in Tokyo, especially Tomonori Hiruta. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

Code Contributions of the Authors. The material presented in the article naturally is based on code and software. In the following we want to make the corresponding contributions of the authors and possible contact points more transparent.

The research described in Section 2.1 was mainly carried out by JW. The topics presented in Section 2.2 are attributed to DW (singleton column stuffing, redundant variable bounds, implied free variables, two-row bound tightening, dual aggregations, matrix module), MW (using GCDs in ranged rows, upgrade to implicit integer variables), and GG (presolving levels). Section 2.3 presents work by GH (distribution diving, revision of diving heuristics), MP (indicator heuristic), and GG (bound, clique, variable bound heuristic). The contributors to Section 2.4 are GH (distribution branching, new reliability notions), GG (branching on multi-aggregated variables), and AMG (improved treatment of nonlinearities). Section 2.5 covers work by BM (edge-concave cuts separator, improved OBBT), FS (upgrade to SOC, improved separation for convex quadratic constraints), and SV (user-defined operators). The research presented in Section 2.6 was mainly performed by TF. The contributors to Section 2.7 are GG (probing, memory management, problem permutation), GH (history transfer, memory management), SM (sparsity information), MM (Python interface, sparsity information, different infinity definitions), MP (memory management, BAK), SV (AMPL interface), RS (Python interface) and BM (feasibility check).

The topics described in Section 3 were treated by AMG (exact solving capability) and MM (sparsity exploitation, further developments). The Windows Makefile for ZIMPL (Section 4) was added by BM. The research presented Section 5 was carried out by YS. Section 6 describes work of JTW (basis cut separator, column pool) and CP (set covering heuristic, refactoring). The work presented in Section 7.1 was mainly done by DR; GG, SJM, TK, YS, and MW contributed to it. Section 7.2 presents research by SS. Finally, Section 7.3 presents work performed by TG.

References

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

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

[3] 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.

[4] T. Achterberg, R. E. Bixby, Z. Gu, E. Rothberg, and D. Weninger. Multi-row presolve reductions in mixed integer programming. In *Proceedings of the Twenty-Sixth RAMP Symposium*, Hosei University Tokyo, 2014.

[5] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, 2004.

[6] T. Achterberg and C. Raack. The MCF-separator – detecting and exploiting multi-commodity flows in MIPs. *Mathematical Programming C*, 2(2):125–165, 2010.

[7] E. D. Andersen and K. D. Andersen. Presolving in linear programming. *Mathematical Programming*, 71:221–245, 1995.

[8] E. M. L. Beale and J. A. Tomlin. Special facilities in general mathematical programming system for non-convex problems using ordered sets of variables. In J. Lawrence, editor, *Proc. 5th International Conference on Operations Research*, pages 447–454. Travistock Publications, London, 1970.

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

[10] S. J. Benson and Y. Ye. Algorithm 875: DSDP5–software for semidefinite programming. *ACM Transactions on Mathematical Software*, 34(4):16:1–16:20, May 2008.

[11] T. Berthold. Primal heuristics for mixed integer programs. Diploma thesis, Technische Universität Berlin, 2006.

[12] T. Berthold. RENS – the optimal rounding. *Mathematical Programming Computation*, 6(1):33–54, 2014.

[13] T. Berthold and D. Salvagnin. Cloud branching. In C. Gomes and M. Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 7874 of *Lecture Notes in Computer Science*, pages 28–43. Springer Berlin Heidelberg, 2013.

[14] R. E. Bixby and E. Rothberg. Progress in computational mixed integer programming–a look back from the other side of the tipping point. *Annals of Operations Research*, 149:37–41, 2007.

[15] P. Bonami, G. Cornuéjols, S. Dash, M. Fischetti, and A. Lodi. Projected chvátal–gomory cuts for mixed integer linear programs. *Mathematical Programming*, 113(2):241–257, 2008.

[16] R. Borndörfer. *Aspects of Set Packing, Partitioning, and Covering.* PhD thesis, TU Berlin, 1998.

[17] A. L. Brearley, G. Mitra, and H. P. Williams. Analysis of mathematical programming problems prior to applying the simplex algorithm. *Mathematical Programming*, 8:54–83, 1975.

[18] A. Caprara and M. Fischetti. {0, 1/2}-chvátal-gomory cuts. *Mathematical Programming*, 74(3):221–235, 1996.

[19] A. Caprara, M. Fischetti, and P. Toth. A heuristic method for the set covering problem. *Operations Research*, 47(5):730–743, 1999.

[20] C. D'Ambrosio, M. Fampa, J. Lee, and S. Vigerske. On a nonconvex MINLP formulation of the Euclidean steiner tree problems in n-space. Technical Report 4528, Optimization Online, 2015.

[21] E. Danna, M. Fenelon, Z. Gu, and R. Wunderling. Generating multiple solutions for mixed integer programming problems. In M. Fischetti and D. P. Williamson, editors, *Integer Programming and Combinatorial Optimization, Proc. of the 12th International IPCO Conference, Ithaca, NY, USA*, volume 4513 of *LNCS*, pages 280–294. Springer, 2007.

[22] E. Danna, E. Rothberg, and C. L. Pape. Exploring relaxation induced neighborhoods to improve MIP solutions. *Mathematical Programming*, 102(1):71–90, 2004.

[23] G. B. Dantzig. Discrete-variable extremum problems. *Operations Research*, 5(2):266–277, 1957.

[24] M. P. de Aragao and R. F. Werneck. On the implementation of MST-based heuristics for the Steiner problem in graphs. In *Proceedings of the 4th International Workshop on Algorithm Engineering and Experiments*, pages 1–15. Springer, 2002.

[25] C. Duin. *Steiner Problems in Graphs.* PhD thesis, University of Amsterdam, 1993.

[26] T. Fischer and M. E. Pfetsch. Branch-and-cut for linear programs with overlapping SOS1 constraints. Technical report, Available on Optimization Online, submitted for publication, 2015.

[27] M. Fischetti and A. Lodi. Local branching. *Mathematical Programming*, 98(1-3):23–47, 2003.

[28] M. Fischetti and A. Lodi. Optimizing over the first chvátal closure. *Mathematical Programming*, 110(1):3–20, 2007.

[29] M. Fischetti and M. Monaci. Branching on nonchimerical fractionalities. *OR Letters*, 40(3):159–164, 2012.

[30] C. A. Floudas. *Deterministic Global Optimization: Theory, Methods and Applications.* Nonconvex Optimization and Its Applications. Springer-Verlag, 2000.

[31] G. Gamrath, T. Berthold, S. Heinz, and M. Winkler. Structure-based primal heuristics for mixed integer programming. In K. Fujisawa, Y. Shinano, and H. Waki, editors, *Optimization in the Real World*, volume 13 of *Mathematics for Industry*, pages 37–53. Springer Japan, 2015.

[32] G. Gamrath, T. Koch, S. J. Maher, D. Rehfeldt, and Y. Shinano. SCIP-Jack – a solver for STP and variants with parallelization extensions. Technical Report 15-27, ZIB, Takustr.7, 14195 Berlin, 2015.

[33] G. Gamrath, T. Koch, A. Martin, M. Miltenberger, and D. Weninger. Progress in presolving for mixed integer programming. *Mathematical Programming Computation*, pages 1–32, 2015.

[34] G. Gamrath, T. Koch, D. Rehfeldt, and Y. Shinano. SCIP-Jack – a massively parallel STP solver. Technical Report 14-35, ZIB, Takustr.7, 14195 Berlin, 2014.

[35] 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, Berlin, Heidelberg, 2010. Springer.

[36] G. Gamrath, A. Melchiori, T. Berthold, A. M. Gleixner, and D. Salvagnin. Branching on multi-aggregated variables. In L. Michel, editor, *Integration of AI and OR Techniques in Constraint Programming, Proc. of CPAIOR*, volume 9075 of *LNCS*, pages 141–156. Springer International Publishing, 2015.

[37] A. M. Gleixner. *Exact and Fast Algorithms for Mixed-Integer Nonlinear Programming.* PhD thesis, Technische Universität Berlin, 2015.

[38] A. M. Gleixner, D. E. Steffy, and K. Wolter. Improving the accuracy of linear programming solvers with iterative refinement. In *ISSAC '12. Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation*, pages 187–194. ACM, July 2012.

[39] A. M. Gleixner, D. E. Steffy, and K. Wolter. Iterative refinement for linear programming. INFORMS Journal on Computing, 2016. To appear. Available as ZIB-Report 15-15, URN:nbn:de:0297-zib-55118.

[40] R. Gomory. An algorithm for integer solutions to linear programming. *Recent Advances in Mathematical Programming*, pages 269–302, 1963.

[41] R. E. Gomory. Solving linear programming problems in integers. *Combinatorial Analysis*, 10:211–215, 1960.

[42] J. A. J. Hall and K. I. M. McKinnon. Hyper-sparsity in the revised simplex method and how to exploit it. *Computational Optimization and Applications*, 32(3):259–283, 2005.

[43] G. Hendel. New rounding and propagation heuristics for mixed integer programming. Bachelor thesis, Technische Universität Berlin, 2011.

[44] G. Hendel. Empirical analysis of solving phases in mixed integer programming. Master's thesis, Technische Universität Berlin, 2014.

[45] G. Hendel. Enhancing MIP branching decisions by using the sample variance of pseudo costs. In *Integration of AI and OR Techniques in Constraint Programming*, volume 9075 of *LNCS*, pages 199–214, 2015. in press.

[46] G. Hendel. Exploiting solving phases for mixed-integer programs. Technical Report 15-64, ZIB, Takustr.7, 14195 Berlin, 2015.

[47] B. Hiller, T. Klug, and J. Witzig. Reoptimization in branch-and-bound algorithms with an application to elevator control. In *Experimental Algorithms, Proc. 12th International Symposium, SEA 2013*, volume 7933 of *LNCS*, pages 378–389, Rome, Italy, 2013. Springer.

[48] J. J. Júdice, H. D. Sherali, I. M. Ribeiro, and A. M. Faustino. A complementarity-based partitioning and disjunctive cut algorithm for mathematical programming problems with equilibrium constraints. *Journal of Global Optimization*, 36(1):89–114, 2006.

[49] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[50] A. Koberstein. *The dual simplex method, techniques for a fast and stable implementation*. PhD thesis, Universität Paderborn, 2005.

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

[52] 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.

[53] T. Koch and A. Martin. Solving Steiner tree problems in graphs to optimality. *Networks*, 32:207–232, 1998.

[54] P. Le Bodic and G. L. Nemhauser. An Abstract Model for Branching and its Application to Mixed Integer Programming. *ArXiv e-prints*, Nov. 2015.

[55] E. L. Lehmann and J. P. Romano. *Testing Statistical Hypotheses*. Springer New York, 2005.

[56] A. N. Letchford and A. Lodi. Strengthening chvátal–gomory cuts and gomory fractional cuts. *Operations Research Letters*, 30(2):74–82, 2002.

[57] M. E. Lübbecke and J. T. Witt. Separation of generic cutting planes in branch-and-price using a basis. In E. Bampis, editor, *Experimental Algorithms – SEA 2015*, volume 9125 of *LNCS*, pages 110–121, Berlin, June 2015. Springer.

[58] H. Marchand. *A polyhedral study of the mixed knapsack set and its use to solve mixed integer programs*. PhD thesis, PhD thesis, Faculté des Sciences Appliquées, Université catholique de Louvain, 1998.

[59] H. Marchand and L. A. Wolsey. Aggregation and mixed integer rounding to solve mips. *Operations research*, 49(3):363–371, 2001.

[60] S. Mars. *Mixed-Integer Semidefinite Programming with an Application to Truss Topology Design*. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2013.

[61] S. Mars and L. Schewe. An SDP-package for SCIP. Technical Report 08/2012, TU Darmstadt and FAU Erlangen-Nürnberg, 2012.

[62] C. A. Meyer and C. A. Floudas. Convex envelopes for edge-concave functions. *Mathematical Programming*, 103(2):207–224, 2005.

[63] R. Misener and C. A. Floudas. Global optimization of mixed-integer quadratically-constrained quadratic programs (MIQCQP) through piecewise-linear and edge-concave relaxations. *Mathematical Programming*, 136(1):155–182, 2012.

[64] R. Misener, J. B. Smadbeck, and C. A. Floudas. Dynamically generated cutting planes for mixed-integer quadratically constrained quadratic programs and their incorporation into GloMIQO 2. *Optimization Methods and Software*, 30(1):215–249, Jan. 2015.

[65] H. Mittelmann. Decision tree for optimization software: Benchmarks for optimization software. http://plato.asu.edu/bench.html.

[66] O. Ozaltin, B. Hunsaker, and T. Ralphs. Visualizing branch-and-bound algorithms. Technical report, Optimization Online, 2007. http://www.optimization-online.org/DB_HTML/2007/09/1785.html. Code available through href="https://projects.coin-or.org/CoinBazaar/wiki/Projects/BAK.

[67] G. Pesant and C.-G. Quimper. Counting solutions of knapsack constraints. In L. Perron and M. A. Trick, editors, *Proc. of CPAIOR*, volume 5015 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 2008.

[68] T. Polzin. *Algorithms for the Steiner problem in networks.* PhD thesis, Saarland University, 2004.

[69] J. Pryor and J. W. Chinneck. Faster integer-feasibility in mixed-integer linear programs by branching to force change. *Computers & Operations Research*, 38(8):1143–1152, 2011.

[70] I. Quesada and I. E. Grossmann. A global optimization algorithm for linear fractional and bilinear programs. *Journal of Global Optimization*, 6:39–76, 1995.

[71] T. K. Ralphs and M. Guzelsoy. Duality and warm starting in integer programming. In *Proceedings of the 2006 NSF Design, Service, and Manufacturing Grantees and Research Conference*, 2006.

[72] T. K. Ralphs and A. Hassanzadeh. On the value function of a mixed integer linear optimization problem and an algorithm for its construction. Technical Report 14T-004, ISE, Lehigh University, 2014.

[73] T. M. Range. An integer cutting-plane procedure for the Dantzig-Wolfe decomposition: Theory. Discussion Papers on Business and Economics 10/2006, Dept. Business and Economics, University of Southern Denmark, 2006.

[74] D. Rehfeldt. A generic approach to solving the Steiner tree problem and variants. Master's thesis, Technische Universität Berlin, 2015.

[75] M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA J. Comput.*, 6(4):445–454, 1994.

[76] A. Saxena, P. Bonami, and J. Lee. Convex relaxations of non-convex mixed integer quadratically constrained programs: extended formulations. *Mathematical Programming*, 124(1):383–411, 2010.

[77] A. Saxena, P. Bonami, and J. Lee. Convex relaxations of non-convex mixed integer quadratically constrained programs: projected formulations. *Mathematical Programming*, 130(2):359–413, 2010.

[78] H. D. Sherali and W. P. Adams. Reformulation–linearization techniques for discrete optimization problems. In M. P. Pardalos, D.-Z. Du, and L. R. Graham, editors, *Handbook of Combinatorial Optimization*, pages 2849–2896. Springer New York, New York, NY, 2013.

[79] H. D. Sherali and B. M. Fraticelli. Enhancing RLT relaxations via a new class of semidefinite cuts. *Journal of Global Optimization*, 22(1-4):233–261, 2002.

[80] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, and T. Koch. ParaSCIP: a parallel extension of SCIP. In C. Bischof, H.-G. Hegering, W. Nagel, and G. Wittum, editors, *Competence in High Performance Computing 2010*, pages 135–148, 2012.

[81] 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. Technical Report ZR 15-53, ZIB, Takustr.7, 14195 Berlin, 2015. Accepted to IPDPS 2016.

[82] Y. Shinano, S. Heinz, S. Vigerske, and M. Winkler. FiberSCIP – a shared memory parallelization of SCIP. Technical Report 13-55, ZIB, Takustr.7, 14195 Berlin, 2013.

[83] F. Tardella. Existence and sum decomposition of vertex polyhedral convex envelopes. *Optimization Letters*, 2(3):363–375, 2008.

[84] E. Uchoa and R. F. Werneck. Fast local search for the steiner problem in graphs. *J. Exp. Algorithmics*, 17:2.2:2.1–2.2:2.22, 2012.

[85] J. Witzig. Reoptimization techniques in MIP solvers. Master's thesis, Technische Universität Berlin, 2014.

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

[87] M. Yamashita, K. Fujisawa, and M. Kojima. Implementation and evaluation of SDPA 6.0 (SemiDefinite Programming Algorithm 6.0). *Optimization Methods and Software*, 18:491–505, 2003.

[88] M. Yamashita, K. Fujisawa, K. Nakata, M. Nakata, M. Fukuda, K. Kobayashi, and K. Goto. A high-performance software package for semidefinite programs: SDPA 7. Technical Report Research Report B-460, Dept. of Mathematical and Computing Science, Tokyo Institute of Technology, September 2010.

[89] MINLP library 2. `http://gamsworld.org/minlp/minlplib2/html/index.html`. revision number r277.

[90] MIPLIB: Mixed Integer Problem LIBrary. `http://miplib.zib.de/`.

[91] Cython. `http://www.cython.org/`.

[92] PEP 0257 – Docstring Conventions. `https://www.python.org/dev/peps/pep-0257/`.

[93] GCG: Generic Column Generation. `http://www.or.rwth-aachen.de/gcg/`.

[94] MOSEK ApS. `http://www.mosek.com/`.

[95] PolySCIP: a solver for multi-criteria integer and multi-criteria linear programs. `http://polyscip.zib.de`.

[96] SCIP: Solving Constraint Integer Programs. `http://scip.zib.de/`.

[97] SCIP-SDP: a mixed integer semidefinite programming plugin for SCIP. `http://www.opt.tu-darmstadt.de/scipsdp/`.

[98] SoPlex: primal and dual simplex algorithm. `http://soplex.zib.de/`.

[99] UG: Ubiquity Generator framework. `http://ug.zib.de/`.

[100] ZIMPL: Zuse Institute Mathematical Programming Language. `http://zimpl.zib.de/`.