AMBROS GLEIXNER, MICHAEL BASTUBBE, LEON EIFLER, TRISTAN GALLY, GERALD GAMRATH, ROBERT LION GOTTWALD, GREGOR HENDEL, CHRISTOPHER HOJNY, THORSTEN KOCH, MARCO E. LÜBBECKE, STEPHEN J. MAHER, MATTHIAS MILTENBERGER, BENJAMIN MÜLLER, MARC E. PFETSCH, CHRISTIAN PUCHERT, DANIEL REHFELDT, FRANZISKA SCHLÖSSER, CHRISTOPH SCHUBERT, FELIPE SERRANO, YUJI SHINANO, JAN MERLIN VIERNICKEL, MATTHIAS WALTER, FABIAN WEGSCHEIDER, JONAS T. WITT, JAKOB WITZIG

# The SCIP Optimization Suite 6.0

ZIB Report 18-26 (June 2018)

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 6.0

Ambros Gleixner[1], Michael Bastubbe[2], Leon Eifler[1], Tristan Gally[3], Gerald Gamrath[1], Robert Lion Gottwald[1], Gregor Hendel[1], Christopher Hojny[3], Thorsten Koch[1], Marco E. Lübbecke[2], Stephen J. Maher[4], Matthias Miltenberger[1], Benjamin Müller[1], Marc E. Pfetsch[3], Christian Puchert[2], Daniel Rehfeldt[1], Franziska Schlösser[1], Christoph Schubert[1], Felipe Serrano[1], Yuji Shinano[1], Jan Merlin Viernickel[1], Fabian Wegscheider[1], Matthias Walter[2], Jonas T. Witt[2], Jakob Witzig[1]

July 2, 2018

**Abstract** The SCIP Optimization Suite provides a collection of software packages for mathematical optimization centered around the constraint integer programming framework SCIP. This paper discusses enhancements and extensions contained in version 6.0 of the SCIP Optimization Suite. Besides performance improvements of the MIP and MINLP core achieved by new primal heuristics and a new selection criterion for cutting planes, one focus of this release are decomposition algorithms. Both SCIP and the automatic decomposition solver GCG now include advanced functionality for performing Benders' decomposition in a generic framework. GCG's detection loop for structured matrices and the coordination of pricing routines for Dantzig-Wolfe decomposition has been significantly revised for greater flexibility. Two SCIP extensions have been added to solve the recursive circle packing problem by a problem-specific column generation scheme and to demonstrate the use of the new Benders' framework for stochastic capacitated facility location. Last, not least, the report presents updates and additions to the other components and extensions of the SCIP Optimization Suite: the LP solver SoPlex, the modeling language Zimpl, the parallelization framework UG, the Steiner tree solver SCIP-Jack, and the mixed-integer semidefinite programming solver SCIP-SDP.

[1] *Zuse Institute Berlin, Department of Mathematical Optimization, Takustr. 7, 14195 Berlin, Germany,* `{gleixner,eifler,gamrath,robert.gottwald,hendel,koch,miltenberger,benjamin.mueller,rehfeldt,schloesser,schubert,serrano,shinano,viernickel,wegscheider,witzig}@zib.de`

[2] *RWTH Aachen University, Chair of Operations Research, Kackertstr. 7, 52072 Aachen, Germany,* `{bastubbe,luebbecke,puchert,walter,witt}@or.rwth-aachen.de`

[3] *Technische Universität Darmstadt, Fachbereich Mathematik, Dolivostr. 15, 64293 Darmstadt, Germany,* `{gally,hojny,pfetsch}@mathematik.tu-darmstadt.de`

[4] *Lancaster University, Department of Management Science, Bailrigg, Lancaster LA1 4YX, United Kingdom,* `s.maher3@lancaster.ac.uk`

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

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

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

## 1 Introduction

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

- the modeling language Zimpl [35],
- the simplex-based linear programming solver SoPlex [68],
- the constraint integer programming solver SCIP [2], which can be used as a fast standalone solver for mixed-integer linear and nonlinear programs and a flexible branch-cut-and-price framework,
- the automatic decomposition solver GCG [21], and
- the UG framework for parallelization of branch-and-bound solvers [59].

All of the five tools can be downloaded in source code and are freely available for usage in non-profit research. They are accompanied by several extensions for more problem-specific classes such as the award-winning Steiner tree solver SCIP-Jack [23] or the mixed-integer semidefinite programming solver SCIP-SDP [20]. This paper describes the new features and enhanced algorithmic components contained in version 6.0 of the SCIP Optimization Suite.

One emphasis of this release lies on new functionality for decomposition methods. Via two newly added plugin types, SCIP now provides a generic framework to solve structured constraint integer programs by Benders' decomposition. This addition complements the existing support for column generation and Dantzig-Wolfe decomposition that has been available from the very beginning through pricer plugins in SCIP and the generic column generation extension of the solver GCG. Furthermore, the new Benders' methods in SCIP have been directly interfaced by GCG such that they can be used conveniently in combination with GCG's automatic structure detection. This interaction provides a good example of the added value that is created by developing and distributing the packages of the SCIP Optimization Suite in a coordinated manner. Another example are the parallel versions of the SCIP extensions SCIP-SDP or SCIP-Jack that have been available via the UG framework since SCIP 5.0.

*Background* From the beginning, SCIP was designed as a branch-cut-and-price framework to solve a generalization of *mixed-integer linear programming* (MIP) called *constraint integer programming* (CIP). MIPs are optimization problems of the form

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

defined by $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, $\ell, u \in \bar{\mathbb{R}}^n$, and the index set of integer variables $\mathcal{I} \subseteq \mathcal{N} := \{1, \dots, n\}$. The usage of $\bar{\mathbb{R}} := \mathbb{R} \cup \{-\infty, \infty\}$ allows for variables that are free or bounded only in one direction. The generalization to CIP was motivated by the modeling flexibility of constraint programming and the algorithmic requirements of integrating it with efficient solution techniques for mixed-integer programming. Specifically, CIPs are optimization problems with arbitrary constraints that statisfy following property: If all integer variables are fixed, the remaining subproblem must form a linear or nonlinear program. This also accommodates for the problem class *mixed-integer nonlinear programming* (MINLP), which next to MIP forms another focus of SCIP's development. MINLPs can be written in the form

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

where the functions $f : \mathbb{R}^n \to \mathbb{R}$ and $g_k : \mathbb{R}^n \to \mathbb{R}$, $k \in \mathcal{M} := \{1, \dots, m\}$, are possibly nonconvex. Within SCIP, we assume that $f$ and $g_k$ are specified explicitly in algebraic form using base expressions that are known to SCIP. The core of SCIP coordinates a central branch-cut-and-price algorithm. Advanced methods can be integrated via predefined callback mechanisms. The solving process is described in more detail by Achterberg [1] and, with focus on the MINLP extensions, by Vigerske and Gleixner [63].

By design, SCIP interacts closely with the other components of the SCIP Optimization Suite. SCIP directly accepts optimization models formulated in Zimpl. Although interfaces to several external LP solvers exist, see also Section 2.7, by default, SCIP relies on SoPlex for solving linear programs (LPs) as a subroutine. As mentioned above, GCG extends SCIP to automatically detect problem structure and generically apply decomposition algorithms based on the Dantzig-Wolfe or the Benders' decomposition scheme. And finally, the default instantiations of the UG framework use SCIP as a base solver in order to perform parallel branch-and-bound in parallel computing environments with shared or distributed memory architectures.

*New Developments and Structure of the Paper* All five packages of the SCIP Optimization Suite 6.0 provide extended functionality. Updates to SCIP are presented in Section 2. The most significant additions and improvements are

- a major extension of the framework's functionality by two new plugin types for performing Benders' decomposition, including an advanced out-of-the-box implementation (Section 2.1),
- two new diving heuristics that interact with conflict information (Sections 2.3.1 and 2.3.2),
- a new aggressive multi-level branching rule (Section 2.4),
- a new measure for selecting cutting planes that considers the distance to the incumbent solution (Section 2.5.2), and
- refined timing options for symmetry detection with orbital fixing (Section 2.6).

An overview of the performance improvements for standalone MIP and MINLP is given in Section 2.2. Section 3 describes the updates in the LP solver SoPlex 4.0, which contain inter alia a new aggregation presolver for improved standalone performance.

In addition to the new core features, SCIP 6.0 comes with a new example implementation for stochastic capacitated facility location, which makes use of the Benders' decomposition framework (Section 4.2). The newly added application Ringpacking implements an advanced column generation scheme based on nonlinear pricing problems for the recursive circle packing problem (Section 4.1). Version 1.3 of the Steiner tree solver SCIP-Jack delivers significant performance improvements for the classical

Steiner tree problem in graphs, the maximum-weight connected subgraph problem, and the prize-collecting Steiner tree problem (Section 4.3).

Section 5 presents version 3.0 of the generic column generation solver GCG, which features a long list of enhancements, most notably

- a full redesign of the automatic structure detection scheme, which now orchestrates multiple detection heuristics dynamically (Section 5.1),
- a restructured pricing scheme providing higher flexibility, in particular regarding heuristic pricers (Section 5.3),
- an interface to SCIP's new Benders' decomposition functionality, turning GCG into a generic Benders' decomposition framework (Section 5.2), and
- many improvements regarding usability (Section 5.4) and technical details of the implementation (Section 5.5).

Updates of the parallelization framework UG are presented in Section 6. UG 0.8.5 comes with a new communication library for shared-memory parallelization based on `C++11` threads, hence improving its portability to non-Unix platforms. Furthermore, users can now specify customized settings to be used during the racing ramp-up phase. This feature has also been used for the parallel version of the mixed-integer semidefinite programming solver SCIP-SDP [20] in order to apply a combination of nonlinear branch-and-bound and an LP-based cutting plane approach.

Finally, note that the modeling language Zimpl in its latest version 3.3.6 is now able to handle sets with more than 2 billion elements due to enhanced data structures.

## 2 Advances in SCIP

### 2.1 A Generic Framework for Benders' Decomposition

*Benders' decomposition* [7] is a popular mathematical programming technique applied to solve large-scale optimization problems. Most commonly, Benders' decomposition is employed to exploit problems with a constraint matrix exhibiting a bordered block diagonal form. This structure is typically observed in stochastic programs and mixed-integer programs that model applications with interconnected resources, such as supply chain management. Problems that are particularly amenable to the application of Benders' decomposition have the form:

$$
\begin{aligned}
\min \quad & c^\top x + d^\top y && (3)\\
\text{s.t.} \quad & Ax \geq b, && (4)\\
& Bx + Dy \geq g, && (5)\\
& x \in \mathbb{Z}^p_+ \times \mathbb{R}^{n-p}_+, && (6)\\
& y \in \mathbb{Z}^q_+ \times \mathbb{R}^{m-q}_+. && (7)
\end{aligned}
$$

The variables $x$ and $y$ are described as the first and second stage variables, respectively. Similarly, the constraints (4)–(5) are the first and second stage constraints, respectively. In many applications, it is possible that the constraint matrix $D$ can be further decomposed into a number of disjoint blocks. In such cases, the problem is described as having a *bordered block diagonal structure*.

Benders' decomposition was originally proposed by Benders [7] as an approach to solve structured problems with a second stage that consists only of continuous variables, i.e., $q = 0$. Since its first development, Benders' decomposition has been extended such that it can be applied to problems where $q > 0$ by employing methods such as the integer cuts proposed by Laporte and Louveaux [37] and Carøe and Tind [12] or Logic-based

Benders' decomposition, see Hooker and Ottosson [29]. In the following, the traditional application of Benders' decomposition will be described. However, the Benders' decomposition framework of SCIP 6.0 provides the capabilities to solve problems with discrete second stage variables.

The application of Benders' decomposition results in the separation of the first and second stage variables and constraints by forming a master problem and subproblem. The subproblem takes a master problem solution $\bar{x}$ as input, forming a problem in the $y$ variable space. For a given solution $\bar{x}$, the Benders' decomposition subproblem is formulated as

$$\begin{aligned} z(\bar{x}) = \min \quad & d^\top y && (8)\\ \text{s.t.} \quad & Dy \geq g - B\bar{x}, && (9)\\ & y \in \mathbb{R}^m_+. && (10) \end{aligned}$$

The dual solutions to (8)–(10) are used to generate classical Benders' optimality and feasibility cuts. An optimal solution to (8)–(10) yields an optimal dual solution $u$ that is used to generate an optimality cut of the form $\varphi \geq u^\top(g - Bx)$, where $\varphi$ is an auxiliary variable added to the master problem as an underestimator of the subproblem optimal objective function value. Similarly, an infeasible instance of (8)–(10), corresponding to an unbounded dual problem, produces a dual ray $v$ that is used to generate a feasibility cut of the form $0 \geq v^\top(g - Bx)$. The optimality cut eliminates a suboptimal master problem solution and the feasibility cut eliminates an infeasible master problem solution, corresponding to $\bar{x}$.

The master problem is formed by the first stage variables and constraints and the cuts generated from solutions to the subproblem. The sets of dual extreme points and rays from (8)–(10) are denoted by $\mathcal{P}$ and $\mathcal{R}$, respectively. Substituting the second stage constraints from the original problem with all optimality and feasibility cuts produces a master problem of the form

$$\begin{aligned} \min \quad & c^\top x + \varphi && && (11)\\ \text{s.t.} \quad & Ax \geq b, && && (12)\\ & \varphi \geq u^\top(g - Bx) && \text{for all } u \in \mathcal{P}, && (13)\\ & 0 \geq v^\top(g - Bx) && \text{for all } v \in \mathcal{R}, && (14)\\ & x \in \mathbb{Z}^p_+ \times \mathbb{R}^{n-p}_+, && && (15)\\ & \varphi \in \mathbb{R}. && && (16) \end{aligned}$$

Since the sets $\mathcal{P}$ and $\mathcal{R}$ are exponential in the size of the input, solving the formulation (11)–(16) containing all optimality and feasibility cuts is computationally impractical. As such, (11)–(16) is relaxed by using subsets $\bar{\mathcal{P}} \subseteq \mathcal{P}$ and $\bar{\mathcal{R}} \subseteq \mathcal{R}$, which are both initially empty. The subproblem is then iteratively solved with candidate master problem solutions to generate cuts to append to $\bar{\mathcal{P}}$ and $\bar{\mathcal{R}}$, and progressively tighten the feasible region. A sketch of the Benders' decomposition solution algorithm is given in Algorithm 1.

There are two methods of implementing Benders' decomposition. The first is to solve the master problem to optimality before evaluating the resulting solution by solving the subproblems and subsequently generating cuts. This algorithm is described as a row generation approach, and is described in Algorithm 1. The second method is to employ Benders' decomposition within a branch-and-cut algorithm. The branch-and-cut approach to Benders' decomposition, which is termed branch-and-check [62], only evaluates solutions by solving the subproblems at nodes where the LP solution is integer feasible. This second approach allows for the Benders' decomposition algorithm to be more integrated with a CIP solver.

**Algorithm 1:** Traditional Benders' Decomposition Algorithm

```
1  $UB \leftarrow \infty$, $LB \leftarrow -\infty$, $\bar{\mathcal{P}} \leftarrow \emptyset, \bar{\mathcal{R}} \leftarrow \emptyset$;
2  while $UB - LB > \epsilon$ do
3      solve (11)–(16), set $(\hat{x}, \hat{\varphi})$ to the solution of MP;
4      $LB \leftarrow c^\top \hat{x} + \hat{\varphi}$;
5      $UB \leftarrow c^\top \hat{x}$;
6      solve (8)–(10) with $\hat{x}$ as input;
7      if (8)–(10) is infeasible then
8          add unbounded dual ray $v$ of (8)–(10) to $\bar{\mathcal{R}}$;
9          $UB \leftarrow \infty$;
10     else
11         $UB \leftarrow UB + z(\hat{x})$;
12         if $z(\hat{x}) > \hat{\varphi}$ then
13             add optimal dual solution $u$ of (8)–(10) to $\bar{\mathcal{P}}$;
```

The possibility to implement the branch-and-check approach to Benders' decomposition has existed within SCIP since its inception. This is due to the integration of constraint programming and integer programming along with the plugin design of the solver. Employing the branch-and-check algorithm using SCIP previously involved the implementation of a constraint handler that managed the solving of the Benders' decomposition subproblems to evaluate candidate solutions from the LP or relaxations and potential incumbent solutions. While previously possible, implementing Benders' decomposition within SCIP still involved an understanding of problem-specific details, especially for the implementation of the Benders' cut generation methods.

For SCIP 6.0, a Benders' decomposition framework has been developed to eliminate much of the implementation effort for the user when employing the algorithm. The framework includes constraint handlers to execute the subproblem solving and cut generation methods at the appropriate points during the branch-and-check algorithm. Further, default subproblem solving and cut generation methods have been provided to simplify the use of the Benders' decomposition algorithm. While the developed framework simplifies the use of the Benders' decomposition algorithm, it still provides the flexibility for the user to develop a custom implementation. In its simplest invocation, the user can employ the Benders' decomposition algorithm to solve a problem by providing an instance in the SMPS format [10]. In its most complex use, within the Benders' decomposition framework a user can implement custom subproblem solving and cut generation methods. The details regarding the implementation and features of the Benders' decomposition framework are provided in the following sections.

### 2.1.1 Usage

There are five different ways in which the Benders' decomposition framework can be used within SCIP. These range from complete automation through to the most flexible approach.

*Using GCG: Automatic Decomposition* The most automated method of using the Benders' decomposition framework is provided in GCG. A Benders' decomposition plugin has been added to GCG and the relaxator has been extended with an additional mode, allowing the user to solve an instance using Benders' decomposition. The structure detection methods of GCG are used to identify the variables and constraints that form the master and subproblems. The subproblems are passed to the Benders' decomposition plugin (`benders_gcg`), so that they are registered with the framework.

When the Benders' decomposition mode is selected, Benders' decomposition is applied to solve all problems provided to GCG—regardless of the problem type. If the appropriate cut generation methods are not available, then the necessary subproblems are merged into the master problem to ensure the instance can be solved. The merging of subproblems is also used if numerical troubles are encountered while solving the master problem.

*Providing an Instance in SMPS Format* SCIP 6.0 has been extended with a collection of readers for the SMPS instance format [10]. The SMPS instance format represents stochastic optimization problems and consists of three file types: a core file (`cor`), a stage file (`tim`), and a stochastic information file (`sto`). Given an instance in the SMPS format, the three files can be provided to SCIP in the previously stated order. Additionally, an `smps` reader has been added that takes a single file containing the paths and filenames of the `cor`, `tim`, and `sto` files, and reads them in the appropriate order.

Providing an instance in SMPS format to SCIP will build the monolithic deterministic equivalent of the stochastic problem by default. Alternatively, the parameter `reading/sto/usebenders` can be set to `TRUE` to employ Benders' decomposition to solve the input stochastic program.

*Using the Default Benders' Decomposition Plugin* The Benders' decomposition plugin `benders_default` is included in SCIP 6.0 as a default plugin. To invoke the default Benders' decomposition plugin, the user creates the SCIP instances for the master problem and the subproblems. The subproblems must contain a copy of the variables from the master problem that will be fixed in the second stage constraints. Most importantly, the names of the master problem variables must be identical in the master and subproblems, since currently a string matching is used to establish the mapping internally. Calling the function `SCIPcreateBendersDefault()` with the master problem, an array of subproblems and the number of subproblems will activate the default Benders' decomposition implementation. In order to execute the Benders' decomposition subproblem solving methods, `cons_benders` must be activated by setting the parameter `constraints/benders/active` to `TRUE`. Additionally, `cons_benderslp` can be activated to employ the two-phase algorithm described below in Section 2.1.4.

*Implementing a Custom Benders' Decomposition Plugin* A custom Benders' decomposition plugin can be implemented by the user to achieve the most flexibility with the framework. Even when implementing a custom Benders' decomposition plugin there are different levels of flexibility. The fundamental callbacks for a Benders' decomposition plugin are the subproblem creation and the variable mapping functions. The subproblem creation method is required to register each subproblem with the Benders' decomposition framework. This is achieved by calling `SCIPcreateBendersSubproblem()`. The variable mapping function is an interface function providing a mapping between the master and subproblem variables. This function takes a variable and an index for the subproblem from which the mapped variable is desired ($-1$ for the master problem). This function is used within the subproblem setup function and the cut generation methods.

Further flexibility is afforded through the subproblem solving and the pre- and post-subproblem callback functions.

*Using Benders' Decomposition through* PySCIPOpt Finally, also the Python interface PySCIPOpt has been extended to include a set of interface functions to the Benders' decomposition framework. The example `flp-benders.py` has been included to demonstrate how to apply the default Benders' decomposition implementation. Additionally, the set of available plugins in PySCIPOpt has been extended to include the Benders' decomposition plugin type. This gives the user the flexibility of implementing a custom Benders' decomposition plugin using using Python instead of the C API.

### 2.1.2 Implementation

The Benders' decomposition framework available within SCIP is designed to provide a flexible platform for using and implementing the Benders' decomposition algorithm. Traditionally, the fundamental components of solving the subproblems and generating Benders' cuts required a problem-specific implementation by the user. The framework provided within SCIP 6.0 aims to reduce the amount of effort required by the user when employing Benders' decomposition.

SCIP has been extended with two new plugin types that provide the functionality for executing the above two critical algorithmic stages. The first plugin type is a *Benders' decomposition plugin* that provides callback functions to allow the user to interact with the subproblem solving loop and cut generation. The fundamental callbacks for a Benders' decomposition plugin are

– the subproblem creation callback, which is used to register the subproblems with the Benders' decomposition framework, and

– a mapping function between the master and subproblem variables, which is called when setting up subproblems with respect to candidate master solutions and generating Benders' cuts.

If no other callbacks are implemented, then the Benders' decomposition framework will automatically execute the candidate solution evaluation and cut generation methods. Other callbacks are provided to allow further customization of the Benders' decomposition solving methods. Details of these additional callbacks can be found in the online documentation. This release includes one Benders' decomposition plugin within SCIP (`benders_default`) and one plugin within GCG (`benders_gcg`).

The second plugin type added to SCIP is the *Benders' decomposition cut plugin* This plugin includes an execution method that is called after each subproblem is solved. The solution of the corresponding subproblem can then be used to generate a constraint or cut for addition to the master problem. The Benders' decomposition framework has been designed to allow subproblems that are general CIPs. As such, it must be stated within the Benders' decomposition cut plugin whether the implemented cut generation method is suitable for convex subproblems (and convex CIP relaxations) or general CIPs. The Benders' decomposition cut plugins available in SCIP 6.0 provide methods to construct classical optimality (`benderscut_opt`) and feasibility (`benderscut_feas`) cuts, the integer cuts proposed by Laporte and Louveaux [37] (`benderscut_int`), and no-good cuts (`benderscut_nogood`). The Benders' decomposition cut generation methods currently provided in SCIP 6.0 support problems with continuous variables in the first and second stages, mixed-integer variables in the first stage and continuous variables in the second stage, and binary variables in the first stage and mixed-integer variables in the second stage.

Finally, the interaction between the master problem and the Benders' decomposition framework is provided by two constraint handlers, `cons_benderslp` and `cons_benders`. Both constraint handlers are used to pass LP, relaxation, pseudo, or candidate solutions to the Benders' decomposition subproblems for evaluation. The first constraint handler, `cons_benderslp`, is included to provide the user with the option to employ the *two-phase algorithm* [45]. This is a commonly used algorithm for Benders' decomposition that tries to improve the convergence of Benders' decomposition by first generating cuts for convex relaxations of the master problem. Once the convex relaxation of the master problem has been solved, then cuts are generated from the candidate integer solutions.

Within the branch-and-check approach, the two-phase algorithm is achieved by setting the enforcement priority of the `cons_benderslp` constraint handler greater than that of the integer constraint handler. Thus when this constraint handler is active, all fractional LP, relaxation, and pseudo solutions are evaluated by the Benders' decomposition framework.

By default, `cons_benderslp` is only active at the root node; however, it is possible to use this constraint handler to evaluate fractional solutions at greater depths in the branch-and-bound tree. The second constraint handler, `cons_benders`, is the most important constraint handler for the Benders' decomposition framework and it must be active for an exact solution approach. This constraint handler has a lower enforcement and check priority than the integer constraint handler so that it is only called to evaluate potential incumbent solutions.

#### 2.1.3 Large Neighborhood Benders' Search

The Benders' decomposition framework includes an enhancement technique that, to the best of the authors knowledge, is only available within SCIP. The *large neighborhood Benders' search* [41] aims to produce higher quality solutions from large neighborhood search heuristics when employed within the Benders' decomposition algorithm. The development of the large neighborhood Benders' search has been motivated by the enhancements achieved through the integration of Benders' decomposition with Local Branching [56] and Proximity search [11].

Traditionally, when Benders' decomposition is used to solve a problem, the large neighborhood search heuristics of a CIP solver are only applied to the master without any consideration of the constraints transferred to the subproblems. As such, the solutions found by the large neighborhood search heuristics are potentially suboptimal, or even infeasible, for the original problem. It is only at the completion of the large neighborhood search heuristics that the candidate solution is evaluated by solving the Benders' decomposition subproblems. At this point, there is no recourse to rerun the heuristic if the proposed solution is suboptimal or infeasible.

The large neighborhood Benders' search attempts to address this issue of potentially suboptimal, or infeasible, solutions being found by the large neighborhood search heuristics. This is achieved by employing Benders' decomposition to solve the auxiliary problems of large neighborhood search heuristics. Within SCIP, the auxiliary problem is created by copying the master problem and applying restrictions to the feasible region. Since all solutions of the auxiliary problem are feasible for the master problem, it is possible to evaluate every potential incumbent by solving the Benders' decomposition subproblems. Evaluating the potential incumbent solutions during the execution of the large neighborhood search heuristics ensures that only solutions that improve the bound of the original problem are accepted.

#### 2.1.4 Additional Features

*Convex and CIP Solving Functions* Benders' decomposition was originally proposed to solve two-stage problems with continuous second-stage variables [7]. However, it is possible to employ Benders' decomposition to solve problems with general CIPs as second stage problems. In the latter case, it is common to generate Benders' cuts from convex relaxations of CIP subproblems to improve the convergence of the algorithm—this is part of the two-phase algorithm described in Section 2.1.2. To permit the generation of Benders' decomposition cuts from convex relaxations of general CIP subproblems within the Benders' decomposition framework, two subproblem solving callbacks are provided within the Benders' decomposition plugins.

The subproblem solving callbacks are executed during two different steps in the candidate solution evaluation process. The first step solves the convex subproblems and the convex relaxations of subproblems. If no cuts are generated from these subproblems, then the second step solves the CIP subproblems, if any exist. If the default Benders' decomposition plugin is used, then the solving of the convex and CIP subproblems is handled internally. If the user implements a custom Benders' decomposition plugin and desires control over the subproblem solving, then the two subproblem solving functions are provided to enable the generation of cuts from convex subproblems and convex relaxations of CIP subproblems.

*Pre- and Post-Subproblem Solving Callbacks* Additional flexibility in custom Benders' decomposition plugins is provided by the pre- and post-subproblem solving callbacks. The pre-subproblem solving callback allows the user to execute any checks or fast evaluations of the candidate solutions prior to the subproblems being solved. This callback can also be used to execute enhancement techniques that involve using different candidate solutions, such as the Magnanti-Wong method [40].

The post-subproblem solving callback is executed after the subproblems are solved and any required cuts are generated and added to the master problem, but before the subproblems are freed. This callback allows the user to perform any actions that require the solution to the subproblems. An example is building a candidate solution for the original problem, which is what this callback is used for in `benders_gcg`. Also, since this callback is executed at the end of the subproblem solving process, any additional clean-up steps can be executed prior to the subproblems being freed.

*Subproblem Merging* A feature of the Benders' decomposition framework in SCIP that is an improvement over other available general frameworks is the ability to merge the subproblems into the master problem. The merging of subproblems can be required if there are infeasibilities, or suboptimalities, that can not be resolved by the generation of cuts. This could be due to numerical troubles or the unavailability of appropriate cuts for the given problem.

At the end of the subproblem solving process, a list of subproblems that are candidates for merging is collated. This list is partitioned into two parts: priority and normal candidates. The priority candidates are those that *must* be merged to allow SCIP to continue solving the instance. An example of a priority merge candidate is a subproblem $s$ that fails to generate a cut due to numerical troubles and it is the only subproblem that is not optimal in the current iteration. In this case, since no cut is generated for any other subproblem, it is not possible to eliminate the current master problem solution causing the suboptimality in subproblem $s$. An example of a normal merge candidate is where the appropriate cut generation methods are not available for the subproblem type, but cuts have been generated for other subproblems.

The merging of subproblems can be performed by calling the API function `SCIPmergeBendersSubproblemIntoMaster()`. The merging process involves transferring all variables and constraints from the selected subproblem to the master problem. If it is not possible to resolve infeasibilities or suboptimalities due to the lack of appropriate cut generation methods, then it is required to merge at least one subproblem. The transferring of all subproblem variables and constraints to the master problem effectively eliminates the current candidate solution.

*Presolving* A presolving step is included within `cons_benders` to compute a lower bound on the auxiliary variables. The lower bound for subproblem $s$ is computed by solving $s$ without fixing any of the master problem variables. If the subproblem is a CIP, then only the root node relaxation is solved. In subproblem $s$, the objective coefficients of the master problem variables are set to zero. As such, the objective function value from this solve is a valid lower bound on the auxiliary variable associated with $s$. To enable this presolving step for Benders' decomposition, the parameter `constraints/benders/maxprerounds` must be set to 1.

*Multiple Decompositions* Another feature of the Benders' decomposition in SCIP that is not available in other general frameworks is the ability to employ multiple decompositions. While it is most common to perform a single decomposition, there are cases where it is useful to use alternate decompositions within one algorithm. An example is if two different subproblem solving methods are desired, such as the compact formulation and using column generation. Additionally, if a tighter relaxation exists, but is more time consuming to solve, it may be desired to only use the associated decomposition less frequently.

Within `cons_benders` and `cons_benderslp`, the subproblem solving methods for each decomposition are executed in decreasing order of priority. If a cut is generated in a decomposition, then no other decomposition will be executed. The lowest priority decomposition will only be called when no cut is generated in all other decompositions.

*Extensibility* Due to the plugin nature of the Benders' decomposition framework, it is easily extended with alternative cut generation methods and enhancement techniques. Additional cut generation methods are added by implementing new Benders' decomposition cut plugins. Enhancement techniques can be implemented through the use of the pre- and post-subproblem solving callback functions.

### 2.2 Overall Performance Improvements for MIP and MINLP

The standalone performance of SCIP for solving mixed-integer linear and nonlinear programs out-of-the-box is an important foundation for most of its algorithmic extensions. This section summarizes the overall progress of the MIP and MINLP core since the last major version SCIP 5.0, which was released December 2017.

#### 2.2.1 Experimental Setup

The diversity of MIP and MINLP and the performance variability of state-of-the-art solvers asks for a careful methodology when measuring performance differences between solver versions. The experimental setup used during SCIP development is described in detail in the release report for SCIP 5.0 [25]. A quick overview is given in the following.

The base testset for MIP evaluation consists of 666 instances compiled from the publicly available instances of the COR@L testset [14] and the five MIPLIB versions [36], excluding instances identified as duplicates or marked as "numerically unstable". For MINLP, 143 instances were manually selected from MINLPLib2 [46], filtering overrepresented classes and numerically troublesome instances. In order to save computational resources during development, testing is usually restricted to a subset of "solvable" instances by removing all that could not be solved by previous releases nor by selected intermediate development versions with five different random seeds. Currently, these MIP and MINLP testsets contain 425 and 113 instances, respectively. Note that for MINLP, an instance is considered solved when a relative primal-dual gap of 0.0001 is reached; for MIP we use gap limit zero.

Each solver version is run with five different random seed initializations, including seed zero, with which SCIP is released. Every pair of instance and seed is treated as an individual observation, effectively resulting in testset sizes of 2125 MIPs and 565 MINLPs. (Hence, in the discussion of performance results the term "instance" is often used when actually referring to an instance-seed-combination, for example, when

**Table 1:** Performance comparison of SCIP 6 versus SCIP 5 on the MIP testset using five different seeds.

| | | SCIP 6.0.0+SoPlex 4.0.0 | | | SCIP 5.0.0+SoPlex 3.1.0 | | | relative | |
|---|---|---|---|---|---|---|---|---|---|
| Subset | instances | solved | time | nodes | solved | time | nodes | time | nodes |
| all | 2113 | 1925 | 76.8 | 1598 | 1914 | 83.0 | 1787 | 1.08 | 1.12 |
| affected | 1786 | 1748 | 66.2 | 1479 | 1737 | 72.2 | 1686 | 1.09 | 1.14 |
| [0,7200] | 1963 | 1925 | 54.0 | 1148 | 1914 | 58.8 | 1295 | 1.09 | 1.13 |
| [1,7200] | 1731 | 1693 | 87.9 | 1594 | 1682 | 96.7 | 1833 | 1.10 | 1.15 |
| [10,7200] | 1402 | 1364 | 180.0 | 2755 | 1353 | 203.6 | 3255 | 1.13 | 1.18 |
| [100,7200] | 875 | 837 | 562.1 | 5630 | 826 | 664.5 | 6798 | 1.18 | 1.21 |
| [1000,7200] | 374 | 336 | 1934.4 | 21472 | 325 | 2312.8 | 26163 | 1.20 | 1.22 |
| diff-timeout | 87 | 49 | 3007.4 | 28229 | 38 | 4055.3 | 36284 | 1.35 | 1.29 |
| both-solved | 1876 | 1876 | 44.7 | 980 | 1876 | 48.2 | 1099 | 1.08 | 1.12 |
| MIPLIBs | 958 | 868 | 98.2 | 2560 | 866 | 101.4 | 2736 | 1.03 | 1.07 |
| COR@L | 1230 | 1112 | 71.1 | 1252 | 1106 | 79.0 | 1435 | 1.11 | 1.15 |

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

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

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

As can be seen in Tables 1 and 2, these statistics are displayed for several subsets of instances. The subset "affected" filters for instances where solvers show differing number of dual simplex iterations. The brackets $[t, T]$ collect the subsets of instances which were solved by at least one solver and for which the maximum solving time (among both solver versions) is at least $t$ seconds and at most $T$ seconds, where $T$ is usually equal to the time limit. With increasing $t$, this provides a hierarchy of subsets of increasing difficulty. The subsets "both-solved" and "diff-timeout" contain the instances that can be solved by both of the versions and by exactly one of the versions, respectively. Additionally, MIP results are compared for the subsets of MIPLIB and COR@L instances, which have a small overlap; MINLP results are reported for the subsets of MINLPs containing "integer" variables and purely "continuous" NLPs.

The experiments were performed on a cluster of computing nodes equipped with Intel Xeon Gold 5122 CPUs with 3.6 GHz and 92 GB main memory. Both versions of SCIP were built with GCC 5.4 and use SoPlex as underlying LP solver: version 3.1.0 (released with SCIP 5.0) and version 4.0.0 (released with SCIP 6.0). Further external software packages linked to SCIP include the NLP solver Ipopt 3.12.5 [32] built with linear algebra package MUMPS 4.10 [4], the algorithmic differentiation code CppAD [13] (version 20160000.1 for SCIP 5.0 and version 20180000.0 for SCIP 6.0), and the graph automorphism package bliss 0.73 [33] for detecting MIP symmetry. The time limit was set to 7200 seconds for MIP and to 3600 seconds for the MINLP runs.

### 2.2.2 MIP Performance

Table 1 analyzes the MIP performance of SCIP 6.0 in comparison to the previous version SCIP 5.0. Despite a brief development period since the last major release in December 2017, it can be seen that notable improvements have been achieved. Overall, SCIP 6 is about 8% faster than SCIP 5. While only a smaller speedup of 3% can be seen on the MIPLIB sets, the impact on COR@L is more pronounced, with 11%. On the subset of harder instances in the [100,7200] bracket, SCIP 6 is even more than 18% faster.

**Table 2:** Performance comparison of SCIP 6 versus SCIP 5 on the MINLP testset using five different seeds.

| | | SCIP 6.0.0+SoPlex 4.0.0 | | | SCIP 5.0.0+SoPlex 3.1.0 | | | relative | |
|---|---|---|---|---|---|---|---|---|---|
| Subset | instances | solved | time | nodes | solved | time | nodes | time | nodes |
| all | 561 | 484 | 143.0 | 18829 | 453 | 176.4 | 20224 | 1.23 | 1.07 |
| affected | 486 | 474 | 92.3 | 15338 | 443 | 117.8 | 16963 | 1.28 | 1.11 |
| [0,3600] | 496 | 484 | 93.4 | 13849 | 453 | 118.5 | 15286 | 1.27 | 1.10 |
| [1,3600] | 481 | 469 | 106.4 | 15951 | 438 | 136.0 | 17560 | 1.28 | 1.10 |
| [10,3600] | 434 | 422 | 147.6 | 20657 | 391 | 190.5 | 22972 | 1.29 | 1.11 |
| [100,3600] | 290 | 278 | 327.3 | 42569 | 247 | 540.3 | 52694 | 1.65 | 1.24 |
| [1000,3600] | 112 | 100 | 550.8 | 91789 | 69 | 1640.0 | 152565 | 2.98 | 1.66 |
| diff-timeout | 55 | 43 | 367.4 | 64193 | 12 | 2662.1 | 237382 | 7.25 | 3.70 |
| both-solved | 441 | 441 | 78.7 | 11429 | 441 | 80.2 | 10837 | 1.02 | 0.95 |
| continuous | 134 | 104 | 179.7 | 36424 | 96 | 208.7 | 27814 | 1.16 | 0.76 |
| integer | 427 | 380 | 133.0 | 15301 | 357 | 167.3 | 18298 | 1.26 | 1.20 |

While the “diff-timeout” subset shows a larger speedup of 35%, the “both-solved” results make clear that the small increase in the number of solved instances by 11 is not the main source for the average reduction of running time. It predominantly stems from improvements over the majority of instances that are already solved by SCIP 5. The main algorithmic contributors to these results are the new Farkas diving heuristic (Section 2.3.1), the tuned ALNS heuristic, updates in the separation of cutting planes (Section 2.5), in particular the newly introduced directed cutoff distance for improved cut selection, and the refined timing for symmetry detection (Section 2.6).

### 2.2.3 MINLP Performance

While SCIP 6.0 does not come with new MINLP-specific features, the tuning of several parts of the code together with some of the MIP developments notably improved MINLP performance. The bound tightening of quadratic equations has been strengthened in certain cases and cuts for quadratic constraints with nonconvex constraint function, but convex feasible region are now marked to be globally valid when possible. Generally, cuts generated by nonlinear constraint handlers are scaled up more aggressively. The gauge separation for convex quadratic constraints introduced with SCIP 4.0 [42] and the disaggregation of quadratic constraints (controlled by the parameter `constraints/quadratic/maxdisaggrsize`) available since SCIP 5.0 [25] have been deactivated. Both features can be helpful for specific instances, but currently their application seems to deteriorate SCIP’s performance on average.

The comparison to SCIP 5.0 is displayed in Table 2. As can be seen, SCIP 6.0 is about 23% faster overall and even 65% faster on the subset of harder instances in the [100,3600] bracket. The improvement is slightly more pronounced on MINLPs with integer variables, but also for pure NLPs SCIP 6.0 is 16% faster. The results on the “diff-timeout” and “both-solved” subsets reveal that these speedups are mostly due to the notable increase in the number of solved instances by 31, i.e., by more than 5% of the testset size.

## 2.3 Primal Heuristics

SCIP 6.0 comes with two new conflict-driven diving heuristics and some performance changes in the adaptive large neighborhood search heuristic. Compared to SCIP 5.0, ALNS starts more conservatively and initially uses the maximum variable fixing rate for defining the neighborhoods. However, the minimum fixing rate of variables that needs to be achieved to run the heuristic is now adjusted dynamically over time in SCIP 6.0.

The new conflict-driven heuristics combine the concepts of primal heuristics and conflict analysis in two different ways: using primal heuristics to derive conflict information and using conflict information to guide a heuristic.

### 2.3.1 Farkas Diving

Primal heuristics typically aim to find improving solutions. As a side effect, variable statistics and information about infeasible parts of the search tree are collected. In contrast to all other diving heuristics in SCIP 6.0, *Farkas diving* aims to construct infeasible subproblems in order to derive new conflict information. To this end, Farkas diving makes all decisions, i.e., variable selection and determining rounding directions, based on the dual of the current LP. The overall goal is to push the solution of the dual LP relaxation towards a proof of local infeasibility.

Suppose a mixed-integer program is given in the form

$$\min\{c^\top x \,:\, Ax \le b,\ \ell_i \le x_i \le u_i \ \text{ for all } i \in \mathcal{N},\ x_i \in \mathbb{Z} \ \text{ for all } i \in \mathcal{I}\},$$

and consider the LP relaxation of a subproblem defined by local bound vectors $\ell'$ and $u'$. This LP relaxation is primal infeasible if and only if there exists a dual ray $(y, s)$ satisfying

$$y^\top A + s = 0, \tag{17}$$

$$y^\top b + s\{\ell', u'\} > 0. \tag{18}$$

Here, we define $s\{\ell', u'\} := \sum_{i:s_i>0} s_i\ell'_i + \sum_{i:s_i<0} s_i u'_i$, i.e., the minimum activity of $s^\top x$ over $x \in [\ell', u']$. Aggregation with respect to the dual multiplier vector $y$ leads to the valid linear constraint $(y^\top A)x \ge y^\top b$, called a *Farkas constraint*. This constraint can be propagated in order to prove infeasibility subject to $\ell'$ and $u'$. Since version 4.0, SCIP implements the technique of *dual ray analysis* and collects and propagates Farkas constraints during the search [42, 67].

Diving heuristics as they are implemented in SCIP 6.0 follow the diving scheme in Algorithm 2. Let $x^\star$ be an optimal primal LP solution of the current local subproblem and $(y^\star, r^\star)$ be the corresponding optimal solution of its dual LP relaxation

$$\max\{y^\top b + r\{\ell', u'\} : y^\top A + r = c,\ (y, r) \in \mathbb{R}^m_+ \times \mathbb{R}^n\}. \tag{19}$$

Clearly, $(y^\star, r^\star)$ neither satisfies (17) nor (18). However, $(y^\star, r^\star - c)$ satisfies at least (17). In order to push the dual solution towards infeasibility, Farkas diving aims to reduce the violation of (18) when tightening the bounds in Lines 8 and 10 of Algorithm 2. To this end, the violation of (18) can be reduced by tightening the upper (or lower) bound of a variable with positive (or negative) objective coefficient. Hence, for determining the rounding direction in Line 4, Step A, it is sufficient to consider the objective coefficient $c_i$ for every integer variable $i$ with fractional LP solution value $x^\star_i$. In order to construct a Farkas constraint with only a few number of bound tightening steps, Farkas diving prefers variables with the most impact on (18) (cf. Line 5). Therefore, the absolute objective coefficient and the change in the local bound are considered.

Note that this rounding strategy has a primal interpretation: diving towards the *pseudo-solution*. The pseudo-solution is the best possible solution subject to variable bounds only. However, the pseudo-solution is often infeasible because it does not consider constraints. In other words, although the main goal of this heuristic is the construction of infeasibility proofs, if primal solutions are found, they can be expected to be of high quality.

In SCIP 6.0 Farkas diving is enabled by default and called directly at the root node. During the search tree it is only executed if it succeeded to produce a primal feasible

**Algorithm 2:** Generic Diving Procedure

```
Input  : LP solution $x^\star$, rounding function $\phi$, score function $\psi$
Output: Solution candidate $\hat{x}$ or NULL
1  $\hat{x} \leftarrow$ NULL, $\tilde{x} \leftarrow x^\star$;
2  $\mathcal{D} \leftarrow \{j \in \mathcal{I} : \tilde{x}_j \notin \mathbb{Z}\}$;                // diving candidates
3  while  $\hat{x} = NULL$ and $\mathcal{D} \neq \emptyset$ do
4      foreach $i \in \mathcal{D}$ do
           (A) determine rounding direction: $d_j \leftarrow \phi(j)$;
           (B) calculate variable score: $s_j \leftarrow \psi(j)$;
5      select candidate $x_j$ with maximal score $s_j$;
6      update $\mathcal{D} \leftarrow \mathcal{D} \setminus j$;
7      if  $d_j = up$  then
8          $\ell_j \leftarrow \lceil \tilde{x}_j \rceil$;                 // tighten local lower bound
9      else
10         $u_j \leftarrow \lfloor \tilde{x}_j \rfloor$;                 // tighten local upper bound
11     (optional) propagate this bound change;
12     if infeasibility detected then
13         analyze infeasibility, add conflict constraints, perform 1-level backtrack,
             goto Line 5 or 20 if $\mathcal{D} = \emptyset$;
14     (optional) re-solve local LP relaxation;
15     if infeasibility detected then
16         analyze infeasibility, add conflict constraints, perform 1-level backtrack,
             goto Line 5 or 20 if $\mathcal{D} = \emptyset$;
17     update $\tilde{x}$ and $\mathcal{D}$ if LP was resolved;
18     if $\tilde{x}_j \in \mathbb{Z}$ for all $j \in \mathcal{I}$ or $\mathcal{D} = \emptyset$ then
19         $\hat{x} \leftarrow \tilde{x}$;
20 return $\hat{x}$;
```

solution during this first call. When activating the feature, our intermediate performance evaluations using two random seeds for comparison showed a 2% speedup on the overall MIP testset and an increased number of instances that could be solved to optimality.

### 2.3.2 Conflict Diving

A well-established diving heuristic in mixed-integer programming is *coefficient diving* [9]. This heuristic guides the search based on so-called *variable locks* [1]. Variable locks give rise whether a variable can always be rounded without violating a model constraints or whether there exists a certain number of model constraints that might be violated after rounding the variable into a certain direction. Therefore, the number of variable down-locks or up-locks measure the "risk" of becoming infeasible when rounding a variable downwards or upwards.

Usually, the number of variable locks does not change after presolving anymore. Hence, variable locks are a static criterion and may incorporate model constraints that do not lead frequently to bound deductions or are not tight in the LP relaxation.

Since this release, SCIP maintains locks implied by conflict constraints, too. This type of locks are called *conflict locks* and are counted separately from variable locks. SCIP uses an aging scheme and a separate pool to maintain all conflict constraints and to discards those that turned out to be less useful than others. The following observation suggests that conflict locks may measure the "risk" of rounding more accurately.

**Observation 2.1.** *Let $(y^\top A)x \geq y^\top b$ be a conflict constraint (or Farkas constraint) derived from an infeasible LP. If the conflict contributes to the conflict up-locks (or conflict down-locks) of a variable $j$, then there exists at least one (model) constraint that contributes to the variable up-locks (or variable down-locks) of $j$, too.*

SCIP 6.0 adds an implementation for a new heuristics *conflict diving*. In contrast to coefficient diving, conflict diving relies on conflict locks (either solely or in a weighted combination with variable locks) and prefers the more "risky" rounding direction. By default, conflict diving is disabled in SCIP 6.0 because a thorough tuning and performance evaluation still needs to be conducted.

## 2.4 Lookahead Branching

With the current release 6.0, SCIP features a new branching rule called *lookahead branching*. This branching method is based on an idea by Glankwamdee and Linderoth [24], who propose to base the branching decision not only on the predicted dual bounds of potential child nodes, but rather take into account potential grand-child nodes as well, i.e., potential nodes two levels deeper in the tree than the current node.

The implementation in SCIP uses a recursive approach that allows to investigate an arbitrary number of levels in the lookahead procedure. The general scheme is illustrated in Figure 1. Starting from the current problem $P$, for each variable $x_i$ with fractional value $\bar{x}_i$, the two potential sub-problems $P^{i-}$ and $P^{i+}$ are created and the corresponding LPs are solved, resulting in LP solutions $\bar{x}^{i-}$ and $\bar{x}^{i+}$. Based on these LP solutions, another auxiliary branching is performed for each fractional variable and the corresponding LPs are solved. This can be repeated as long as desired, but since the number of LPs to be solved is exponential in the maximum recursion depth, more than two levels are usually too expensive.

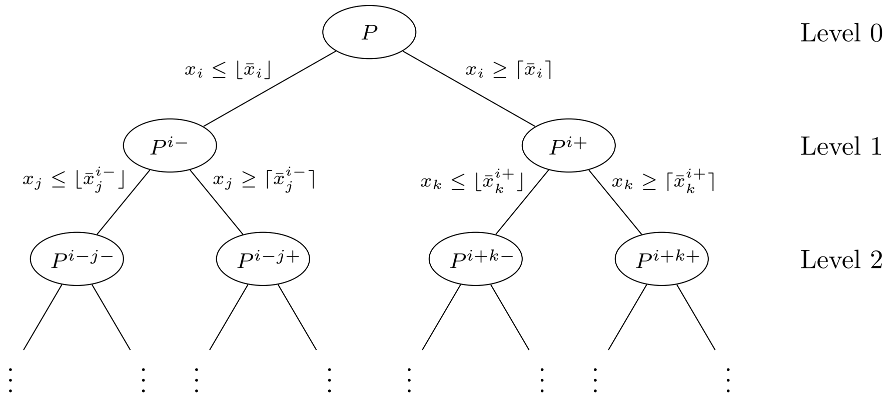
$P$
Level 0
$x_i \leq \lfloor \bar{x}_i \rfloor$
$x_i \geq \lceil \bar{x}_i \rceil$
$P^{i-}$
$P^{i+}$
Level 1
$x_j \leq \lfloor \bar{x}_j^{i-} \rfloor$
$x_j \geq \lceil \bar{x}_j^{i-} \rceil$
$x_k \leq \lfloor \bar{x}_k^{i+} \rfloor$
$x_k \geq \lceil \bar{x}_k^{i+} \rceil$
$P^{i-j-}$
$P^{i-j+}$
$P^{i+k-}$
$P^{i+k+}$
Level 2

**Figure 1:** Illustration of the lookahead branching procedure.

Based on the information provided by these auxiliary sub-trees, a branching decision is taken at the original level. This is done mainly based on the dual bounds of the auxiliary nodes, but rather than combining just two dual bounds to one score as for strong branching, many more dual bounds from deeper levels are taken into account. Here, the SCIP implementation uses the dual bounds of child nodes in level two and deeper to improve the dual bounds originally computed for their parent node, while the dual bounds of the level one nodes are combined with a product score, as usually done in SCIP [1]. This behavior is different to the proposed method of Glankwamdee and Linderoth but proved to perform better in practice. In the lookahead process, additional information can be extracted, including bound changes, locally valid constraints, feasible solutions, and pseudo cost.

Since the full-scale version of lookahead branching is too time consuming for practical applications, a faster version called *abbreviated lookahead branching* is available. It computes standard strong branching scores for all candidates and performs the expensive lookahead procedure only for the $k$ candidates with the best scores. In deeper levels, again only the $k$ best candidates are considered, re-using strong branching scores computed beforehand.

Computational results with a preliminary version of abbreviated lookahead branching with $k = 4$ showed a node reduction by almost 40 % on all instances of the last three MIPLIB benchmark sets that were solved with some branching within 5 hours. When measuring tree size using the fair node number [22], which takes into account the side-effects of strong branching and lookahead branching, the reduction still amounts to 35 %, which shows that the branching decision that lookahead branching takes are indeed of a higher quality. In the end, a combination of abbreviated lookahead branching and full strong branching, where the former is only applied at the first five levels of the branch-and-bound tree, outperforms standard full strong branching. It solves three more instances within the time limit and leads to a slight speedup. All in all, this method offers a viable alternative in the context of memory-restricted environments or massive parallelization because it reduces the branch-and-bound tree size. For more details, we refer to the Master's thesis of Christoph Schubert [58].

## 2.5 Improvements in Cutting Plane Separation

The use of cutting planes is among the core techniques contributing to the effectiveness of modern MIP solvers [3]. Successfully applying cutting plane techniques computationally requires algorithms and methods for generation, selection, and management of cuts. Version 6.0 of SCIP includes improvements within the separation of complemented mixed-integer rounding (CMIR) cuts [43] and the general cut selection algorithm.

### 2.5.1 The CMIR Separator

The CMIR separation procedure comprises heuristics for aggregating rows, substituting bounds, and generating MIR cuts from the resulting single row relaxation. During the last stage, different scaling factors are tested within the cut generation heuristic and the scaling factor that yields the most efficacious MIR cut is chosen. SCIP 5.0 tries (the inverse of) each nonzero coefficient of integral variables in the single row relaxation as scaling factor, if the variable has a solution value that is far away from its bounds. The reasoning behind this strategy is that the violation of an MIR cut decreases when the coefficients of those variables are rounded. Therefore, it is desirable to scale the single row relaxation such that these coefficients are integral, or almost integral, if this results in a fractional right-hand side. To extend the simple heuristic employed in SCIP 5.0, starting from version 6.0 SCIP tries to find the smallest scaling factor that makes all these coefficients integral by computing the greatest common divisor of the denominators of the coefficients. If the right-hand side remains fractional this scaling factor is considered in addition to the ones already tested in SCIP 5.0.

### 2.5.2 Directed Cutoff Distance: A New Measure for Cut Selection

As has been pointed out in the recent survey of Dey and Molinaro [16], the selection of cutting planes is a challenging problem that is still not well understood. Usually it is desired to maximize the dual bound gain achieved by the selected set of cuts, while avoiding to clutter the LP relaxation with too many useless cuts. The best dual bound clearly is achieved by adding all cuts to the LP relaxation, but commonly only a small subset of them will be active at the optimal solution after reoptimizing the LP. Moreover, a largely increased size of the LP and the occurrence of many parallel cuts is likely to affect the numerical stability and the solving time of the LP adversely. Therefore, adding all cuts to the LP increases the solving time despite reducing the number of branch-and-bound nodes for most instances.

Most solvers employ a heuristic approach to select the set of cutting planes added to the LP relaxation. Successful methods described in the literature [2, 65, 5] commonly use a scoring function for assessing the quality of cutting planes and the parallelism between them to measure their similarity. A greedy approach that selects the cut with the highest score and discards similar cuts is then employed iteratively until no more candidates are left, or the maximum amount of cuts has been selected. This general algorithm is customized by the choice of the scoring function and the threshold for the maximum parallelism between cuts. In order to compute meaningful scores for the cutting planes it is necessary to compute some kind of measure that indicates the quality of a cut. For the purpose of cut selection, however, it is unclear what constitutes the quality of an individual cut due their interaction.

Among other measures, SCIP 5.0 uses the *efficacy*, sometimes also called *cutoff distance*: the Euclidean distance between the half-space defined by the cut and the current LP solution. The efficacy, however, can be small for cuts which are considered "strong" in some other sense, for instance, because they are facets of the convex hull of integer solutions. Version 6.0 of SCIP introduces a new measure that can overcome these problems in some cases, and is still cheap to compute.

The idea of the new measure is to use the cutoff distance in a more relevant direction, instead of using the shortest distance to the half-space of the cut. A relevant direction should point towards the feasible region. Often points that are within the integer polytope are found early on by primal heuristics. Hence, the direction from the current LP solution towards the current incumbent solution is readily available in many cases. In these cases, the distance between the current LP solution and the cut along the segment that joins the current LP and incumbent solutions can be computed easily and is used as part of the score in SCIP 6.0. We call this measure the *directed cutoff distance*.

Formally, given a cut $a^\top x \leq b$, the current LP solution $\tilde{x}$, and the current incumbent solution $\bar{x}$, let $d = \frac{\bar{x}-\tilde{x}}{||\bar{x}-\tilde{x}||}$. Then the directed cutoff distance is given by

$$\frac{a^T \tilde{x} - b}{|a^\top d|}.$$

Since $d$, the normalized direction from $\tilde{x}$ towards $\bar{x}$, only needs to be computed once when separating a fixed $\tilde{x}$, the computational effort is comparable to computing the efficacy. The weight of the directed cutoff distance in the linear combination used to compute the score of a cut is adjusted via the parameter `separating/dircutoffdistfac`. The default setting in SCIP 6.0 uses the weight 0.5 in addition to the existing weights for the other measures: the efficacy (default weight 1.0), the integral support (default weight 0.1), and the parallelism with the objective function (default weight 0.1). At the time of activating this feature, this gave a speed-up of 4% on all instances and 9% on harder instances in the [100,7200] bracket.

## 2.6 Improvements in Symmetry Handling

Symmetries in mixed-integer programs typically have an adverse effect on the running time of branch-and-bound procedures because symmetric solutions are explored repeatedly without providing new information to the solver. To handle symmetries on binary variables, two symmetry handling approaches have been implemented and are available in SCIP since version 5.0: a pure propagation approach, so-called *orbital fixing* [44, 47, 48], and a separation-based approach via so-called *symretopes* [28]. In either approach, the user has the possibility to use the symmetries of the original or the presolved problem as the basis for symmetry reductions.

With the release of SCIP 6.0, the timing scheme for computing symmetries has been refined for the orbital fixing approach. Via the parameter

`propagating/orbitalfixing/symcomptiming`

the user can control if symmetries are computed before presolving (value 0), at the end of presolving (value 1), or at the end of processing the root node (value 2), which is also the default value of the parameter. The reason for this is that symmetries typically can be computed very fast after the reductions at the root node. Also computing symmetries after the root node has the advantage that symmetry handling cannot change the solution process on very easy instances that can be solved within the root. Further, SCIP 6.0 allows to handle symmetry via orbital fixing already during presolving by setting parameter `propagating/orbitalfixing/performpresolving` to `TRUE`.

Moreover, in the previous implementation it was not possible to update symmetry information during the solving process. To add more flexibility in symmetry handling, the method `SCIPgetGeneratorsSymmetry()` has been extended by an additional argument to allow for recomputing symmetries of the problem. For example, it is now possible to use orbital fixing after a restart of the solution process occured, by setting `propagating/orbitalfixing/enabledafterrestarts` to `TRUE`. In addition, if a user writes her own symmetry handling plugin, she can access the symmetries of the subproblem at the current branch-and-bound node by recomputing symmetries.

## 2.7 Updates in the Linear Programming Interfaces

SCIP allows to be interfaced with several LP solvers: Clp[1], CPLEX[2], Gurobi[3], MOSEK[4], QSopt[5], SoPlex, and Xpress[6]. In SCIP 6.0, the corresponding Linear Programming Interfaces (LPIs) have been updated as follows. The documentation of features and functions has been made more precise. Several checks for wrong usage have been added and the extension of internal unit tests during the development allowed to fix several minor bugs. For example, the LPI for the open-source solver Clp has been improved and is now much more stable for recent versions of Clp. Finally, the interface has been tuned for several solvers (Gurobi, MOSEK, Xpress), and the SCIP solution process using these solvers is now quite stable.

## 2.8 Technical Improvements and Interfaces

A set of smaller technical changes and improvements have been performed with SCIP 6.0, detailed in the following.

### 2.8.1 Generalized Variable Locks

SCIP uses the concept of *variable locks* in order to count, for each variable, the number of constraints that may become infeasible when increasing or decreasing the value of this

[1]`projects.coin-or.org/Clp`
[2]`www.ibm.com/analytics/cplex-optimizer`
[3]`www.gurobi.com/`
[4]`www.mosek.com`
[5]`https://www.math.uwaterloo.ca/~bico/qsopt/`
[6]`http://www.fico.com/en/products/fico-xpress-optimization`

variable in a solution. This generalizes the information given by the signs of coefficients in the matrix representation of a mixed-integer program to constraint integer programs [1]. Until SCIP 5.0, these variable locks were only counted for model constraints having their "check" flag set to true. SCIP 6.0 extends the concept of variable locks and introduces lock types. The new *conflict locks* regard constraints in the conflict pool, while the classical locks are now captured in the *model locks*. The main motivation for this generalization was the work on the new conflict-driven diving heuristics described in Section 2.3. The conflict diving heuristic uses a diving scheme similar to coefficient diving, but instead of taking the fixing decision based on model locks, it uses conflict locks or a combination of both lock types.

### 2.8.2 Checks and Statistics regarding LP

Analogous to the previously existing checks of primal and dual feasibility of LP solutions, SCIP 6.0 now double-checks the feasibility of Farkas rays returned by the LP solver. The check is controlled by the new parameter `lp/checkfarkas`, which is set to true by default.

In addition, the statistics now report the number of additional LP solves that were triggered because the initial solution returned by the LP solver was marked as instable. Both features help to better detect and deal with numerical instability related to LP solving.

### 2.8.3 Support for Nonlinear Constraint Functions in PySCIPOpt

The Python interface PySCIPOpt available and developed at `https://github.com/SCIP-Interfaces/PySCIPOpt` now supports a larger set of nonlinear functions. Previously, the only nonlinear expressions supported were polynomials. With the new version, PySCIPOpt models may include: non-integer exponents, logarithms, exponentials, absolute values, square roots, and divisions. An example of these new functions can be found in `tests/test_nonlinear.py`.

### 2.8.4 Further Changes

The order for checking constraint handler feasibility of solutions in the original problem has been modified. Constraint handlers with negative check priority that do not need constraints are now checked only after all other constraint handlers.

Furthermore, the number of calls to presolvers as controlled by parameters named `.../maxrounds` and `.../maxprerounds` is now limited by the number of rounds that a presolving step has actually been executed, not (like previously) by the total number of presolving steps performed so far. This simplifies tuning of different presolving steps and reduces random side effects between presolvers.

Finally, the large source file `scip/scip.c` has been split into several smaller implementation files `scip/scip_*.c` for improving the accessibility of the code. The file `scip/scip.c` was removed. This does not affect external SCIP projects as the central header file `scip/scip.h` still remains the standard include for API use.

## 3 SoPlex

SoPlex 4.0 is a major update on the last version, albeit mostly due to technical changes.

### 3.1 Aggregation Presolver

Equations with two variables, i.e., of the form

$$a_1 \cdot x_1 + a_2 \cdot x_2 = b \tag{20}$$

are now removed by aggregating either $x_1 = (b - a_2 \cdot x_2)/a_1$ or $x_2 = (b - a_1 \cdot x_1)/a_2$, depending on the size of the coefficients and the potentially tightened bounds on the variables. This presolving step can decrease the solving time significantly on suitable instances that contain constraints of said type. An example of the possible performance impact is given in Table 3.

**Table 3:** Comparison of presolving reductions and total solving time on instance `sgpf5y6`.

| | cols | rows | time (in seconds) |
|---|---|---|---|
| original instance | 308634 | 246077 | – |
| SoPlex 3.1 | 206033 | 143546 | 718 |
| SoPlex 4.0 | 105453 | 42966 | 22 |

Note that this presolving reduction is already available within SCIP. Hence, this improvement only impacts performance when using SoPlex as a standalone LP solver.

### 3.2 Handling of Numerical Difficulties

SoPlex 4.0 introduces a new solution status `OPTIMAL_UNSCALED_VIOLATIONS` to signal numerical violations that could not be resolved. This is meant to be a last resort when all other options have been exhausted and the last version would have terminated the solving process unsuccessfully. This new status has been integrated into the LP interface of SCIP 6.0 to treat those cases either as optimally solved or not, depending on the parameters in SCIP, namely `lp/checkdualfeastol`, `lp/checkprimalfeastol`, and `lp/checkstability`. A new API method `SoPlex::ignoreUnscaledViolations()` has been implemented to transform the new solution status to `OPTIMAL`.

### 3.3 Technical Improvements

The organization of header files has been changed to enable the inclusion of a single header file `soplex.h` with all other header and source files being moved to a subdirectory `src/soplex`. This avoids name clashes and provides a clean file structure when installing the solver.

Furthermore, there is a new parameter `bool:ensureray` that controls whether SoPlex may skip the generation of a proof for primal or dual infeasibility. This parameter is set to `false` when running SoPlex standalone because the proof is usually not required. It is active within SCIP, though, because this information is used, for instance, to generate conflicts.

Finally, the `LEGACY` mode for compatibility with pre-`C++11` compilers has been removed to simplify code maintenance.

## 4 Applications and Extensions

In addition to the core solvers, the SCIP Optimization Suite is accompanied by several applications and extensions for various classes of mathematical programming problems.

SCIP 6.0 includes the new application RINGPACKING, which exploits SCIP's functionality as a column generation framework for solving the recursive circle packing problem (RCPP) [50, 26]. The CYCLECLUSTERING application [66, 17] has been improved by bugfixes and code refactoring, and the performance of the Steiner tree solver SCIP-JACK [23] has been improved significantly for several problem classes. The package SCIP-SDP [20] for solving mixed-integer semidefinite programs has been updated with bugfixes and an extension of the SDPA-reader to indicator constraints. Furthermore, the parallelized version of SCIP-SDP has been extended by racing parameters to allow a combination of nonlinear branch-and-bound and an LP-based cutting plane approach, see Section 6.

### 4.1 Recursive Ring Packing

The RINGPACKING application implements a column generation algorithm that solves the recursive circle packing problem (RCPP) [50] exactly. Given a set of *ring types* $\mathcal{T} = \{1, \ldots, T\}$, where each $t \in \mathcal{T}$ is associated with an internal radius $r_t \in \mathbb{R}_+$, an external radius $R_t \in \mathbb{R}_+$, and a demand $D_t \in \mathbb{Z}_+$, the objective is to select a minimum number of identical rectangles of size $W \times H$ such that all rings can be packed into these rectangles in a nonoverlapping way. Rings can be put either recursively into larger ones or directly into a rectangle.

The RCPP contains multiple sources of symmetry. Any permutation of rectangles, i.e., relabeling rectangles, constitutes an equivalent solution to RCPP. Even worse, there is also considerable symmetry inside a rectangle. First, rotating or reflecting a rectangle gives an equivalent rectangle since both contain the same set of rings. Second, two rings with same internal and external radius can be exchanged arbitrarily inside a rectangle, again resulting in an equivalent rectangle packing.

The RINGPACKING application implements the techniques developed by Gleixner, Maher, Müller, and Pedroso [26]. First, the concept of circular and rectangular patterns is introduced in order to decompose packings of rectangle packings. Afterward, these patterns are used to apply a Dantzig-Wolfe decomposition [15] to the RCPP in order to break the above mentioned symmetry between equivalent rectangles. A column generation and column enumeration algorithm is used to solve the continuous relaxation of the obtained reformulation.

More precisely, a vector $P \in \mathbb{Z}_+^T$ is a *rectangular pattern* if and only if $P_t$ many circles of each type $t \in \mathcal{T}$ can be packed together into a rectangle at the same time. Here, *circle* corresponds to a ring with zero inner radius and acts as a placeholder for rings of the same external radius. Similarly, a tuple $(t', P) \in \mathcal{T} \times \mathbb{Z}_+^T$ is a *circular pattern* if it is possible to pack $P_t$ many circles of each type $t \in \mathcal{T}$ together into one ring of type $t'$. Let $\mathcal{RP}$ and $\mathcal{CP}$ denote the set of all rectangular or circular patterns, respectively. Figure 2 shows an example of circular and rectangular patterns.

The Dantzig-Wolfe decomposition $PDW(\mathcal{RP})$ of the RCPP reads as

$$
\begin{aligned}
\min \quad & \sum_{P \in \mathcal{RP}} z_P && && \text{(21a)}\\
\text{s.t.} \quad & \sum_{C=(t,P) \in \mathcal{CP}} z_C \geq D_t && \text{for all } t \in \mathcal{T}, && \text{(21b)}\\
& \sum_{C=(t,P) \in \mathcal{CP}} z_C \leq \sum_{P \in \mathcal{RP}} P_t \cdot z_P + \sum_{C=(t',P) \in \mathcal{CP}} P_t \cdot z_C && \text{for all } t \in \mathcal{T}, && \text{(21c)}\\
& z_C \in \mathbb{Z}_+ && \text{for all } C \in \mathcal{CP}, && \text{(21d)}\\
& z_P \in \mathbb{Z}_+ && \text{for all } P \in \mathcal{RP}, && \text{(21e)}
\end{aligned}
$$

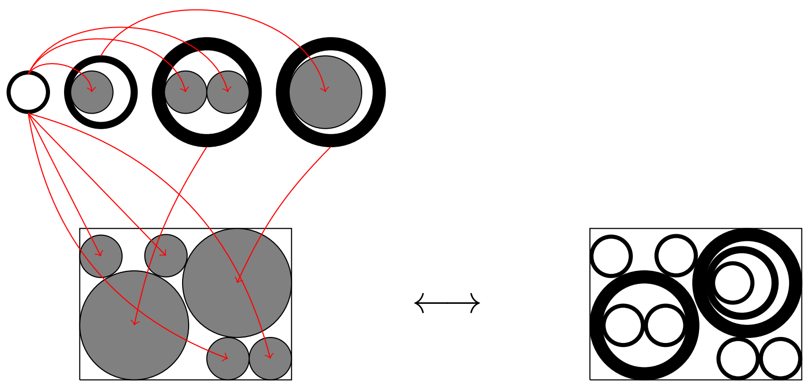

**Figure 2:** An example with four circular patterns and one rectangular pattern showing how patterns are used to model the combinatorial part of RCPP. Each line connects a circular pattern to a circle. The number of outgoing edges is equal to the number of rings that are used. The corresponding packing is shown on the right.

where $z_P$ and $z_C$ is the number of used rectangular patterns $P \in \mathcal{RP}$ and circular patterns $C \in \mathcal{CP}$, respectively. The objective (21a) minimizes the total number of used rectangles. Constraint (21b) ensures that the demand for each ring type is satisfied. The recursive decisions how to place rings into each other are implicitly modeled by (21c). Each selection of a pattern $(t', P) \in \mathcal{CP}$ or $P \in \mathcal{RP}$ enables to use $P_t$ circular patterns of type $t$.

The reformulation is solved by applying two different methods to handle the exponential number of variables. On the one hand, the implementation contains a column *enumeration* algorithm to compute all (relevant) circular patterns. On the other hand, a column *generation* approach that dynamically generates rectangular pattern variables is used in order to solve the LP relaxation of (21). The two kinds of variables are treated differently since in typical applications the rectangles are quite large compared to the rings. Therefore, $\mathcal{RP}$ is expected to be much larger than $\mathcal{CP}$, making an enumeration intractable.

The main step of the circular pattern enumeration algorithm is to verify whether a given tuple $(t, P) \in \mathcal{T} \times \mathbb{Z}_+^{\mathcal{T}}$ is in the set $\mathcal{CP}$ or not. A tuple can be checked by solving the following nonlinear nonconvex verification problem:

$$\left\| \begin{pmatrix} x_i \\ y_i \end{pmatrix} - \begin{pmatrix} x_j \\ y_j \end{pmatrix} \right\|_2 \geq R_i + R_j \qquad \text{for all } i, j \in C, i < j, \tag{22a}$$

$$\left\| \begin{pmatrix} x_i \\ y_i \end{pmatrix} \right\|_2 \leq r_t - R_i \qquad \text{for all } i \in C, \tag{22b}$$

$$x_i, y_i \in \mathbb{R} \qquad \text{for all } i \in C. \tag{22c}$$

Here, $C$ is the index set of individual circles, so if $P_\tau = k$ for a circle type $\tau$, $C$ contains the indices of $k$ many circles of type $\tau$. Furthermore, $(x_i, y_i)$ is the corresponding center and $R_i$ the external radius of a circle $i \in C$. The model checks whether all circles can be placed in a nonoverlapping way into a ring of type $t \in \mathcal{T}$. Constraint (22a) ensures that no two circles overlap, and (22b) guarantees that all circles are placed inside a ring of type $t$. Symmetry handling constraints and a dominance relation between circular patterns are introduced in order to solve this problem more efficiently. Additionally, a greedy heuristic is used to verify simple patterns before solving (22).

Concerning rectangular patterns, the LP relaxation of $PDW(\mathcal{RP}')$ is called the restricted master problem of (21) for a subset $\mathcal{RP}' \subseteq \mathcal{RP}$. Let $\lambda \in \mathbb{R}^{\mathcal{T}}_{+}$ be the nonnegative vector of dual multipliers for (21c) after solving the LP relaxation of $PDW(\mathcal{RP}')$ for the current set of rectangular patterns $\mathcal{RP}'$. To compute a rectangular pattern with the most negative reduced cost the following pricing problem has to be solved:

$$\min_{P \in \mathcal{RP}} \left\{ 1 - \sum_{t \in \mathcal{T}} \lambda_t P_t \right\}. \tag{23}$$

This can be modeled as the maximization variant of the circle packing problem for a single rectangle. This problem is known to be $\mathcal{NP}$-hard [38] and solved by applying a sub-SCIP to a nonconvex MINLP formulation. Greedy heuristics are implemented to find improving columns quickly during the early pricing calls, and to avoid solving the expensive MINLP pricing problems. Still, for hard instances the pricing problems at the root node cannot be solved to optimality. In this case the implementation resorts to price-and-branch: a last valid dual bound is derived using Farley's Lemma [18], column generation is terminated, and the remaining MIP is solved.

To address the potentially expensive enumeration of circular patterns, [26] presents a *price-and-verify* algorithm, employing a dynamic verification process. Instead of enumerating *all* circular patterns at the start, strict working limits are applied when solving (22) to obtain an initial set of circular patterns; this initial set may contain both feasible patterns and patterns for which neither feasibility nor infeasibility could be verified within these limits. After the pricing problem has been solved, only those circular patterns which are relevant in the LP solution of the restricted master problem are then verified. If one turns out to be infeasible, the pricing loop is entered again with this new information. Algorithm 3 gives a detailed description of this procedure.

**Algorithm 3:** Price-and-verify

```
   in  : internal and external radii $r$ and $R$, demands $D$
   out: LP solution $z^*$ of the master problem
1  $(\mathcal{CP}_{feas}, \mathcal{CP}_{unknown}) \leftarrow$ EnumeratePatterns$(r, R, D)$;      // initial verification
2  $\Psi_C \leftarrow 0$ for all $C \in \mathcal{CP}_{unknown}$;
3  while $\exists R \in \mathcal{RP} : red_R < 0$ do
4      $\mathcal{RP} \leftarrow \mathcal{RP} \cup \{R\}$;                                  // pricing loop
5  $z^* \leftarrow$ solve $LP(RMP)$;
6  while $\exists C \in \mathcal{CP}_{unknown} : z^*_C > 0 \wedge \Psi_C = 0$ do
7      $status \leftarrow$ solve verification NLP (22);                  // verification step
8      $\Psi_C \leftarrow 1$;
9      if $status =$ "feasible" then
10         $\mathcal{CP}_{unknown} \leftarrow \mathcal{CP}_{unknown} \setminus \{C\}$;
11         $\mathcal{CP}_{feas} \leftarrow \mathcal{CP}_{feas} \cup \{C\}$;
12     if $status =$ "infeasible" then
13         fix $z_C \leftarrow 0$;
14         $\mathcal{CP}_{unknown} \leftarrow \mathcal{CP}_{unknown} \setminus \{C\}$;
15         go to 3;                                         // enter pricing loop again
16 if $\exists C \in \mathcal{CP}_{unknown} : z^*_C > 0$ then
17     fix $z_C \leftarrow 0$ for all $C \in \mathcal{CP}_{unknown}$;
18     $\mathcal{CP}_{unknown} \leftarrow \emptyset$;
19     go to 3;                  // mark dual bound as invalid and continue pricing
20 return $z^*$;
```

First, in Line 1, the column enumeration algorithm for circular patterns is used with strict working limits in order to produce an initial set $(\mathcal{CP}_{feas}, \mathcal{CP}_{unknown})$ of circular patterns some of which are verified feasible and some of which have not been verified yet. Afterwards, in Lines 3 to 5, the pricing problem and then the LP relaxation of the restricted master problem are solved as described above. If the LP solution contains a variable $z^*_C > 0$ corresponding to an unverified circular pattern $C$, the verification algorithm is called again, see Line 7, with higher working limits than in the beginning.

There are three possible outcomes. If the verification is successful and the pattern turns out to be feasible, see Line 9, or if the verification process failed by reaching the working limits, the algorithm looks for another variable satisfying the above conditions. If, however, the pattern could be proven to be infeasible, the variable is fixed to zero and the pricing loop is entered again, see Lines 12 to 15. Of course, if the verification process failed, in further iterations it should not be called again for the same pattern. This is handled by the variable $\Psi$. It can happen that none of the circular patterns with positive LP solution value can be verified. In that case, all unknown patterns are discarded and the master LP seizes to provide a valid dual bound for RCPP, see Lines 16 to 19. As in the case of prematurely terminated pricing, a last valid dual bound is reported and the solution process enters a restricted price-and-branch phase.

The advantage of dynamically verifying circular patterns is that the information gained from the LP solution of the restricted master problem is used to perform expensive verification only for relevant patterns. Once the pricing loop returns an LP solution with $z^*_C = 0$ for all unverified circular patterns $C \in \mathcal{CP}$, the LP relaxation of (21) has been solved for the root node. If there exist any integer variables with fractional solution value, the solving process continues with branching. Further details and a thorough computational evaluation can be found in [26]. Experimental results on a large test set show that the implementation method not only succeeds in computing exact, sometimes optimal dual bounds, but can even produce primal solutions that improve upon those computed by dedicated heuristics from the literature.

### 4.2 A Benders' Decomposition Example for Stochastic Capacitated Facility Location

The *stochastic capacitated facility location problem* (SCFLP) is included in SCIP 6.0 to provide an example of using the Benders' decomposition framework (see Section 2.1). The formulation of the SCFLP used for this example has been adapted from the model developed by Louveaux [39]. The first stage consist of selecting the set of facilities to open and the second stage attempts to satisfy the customer demand—in a set of scenarios—from the open facilities. The sets $I$ and $J$ denote the facilities and customers respectively. The scenarios for this problem describe different demand profiles across the set of customers. The set of scenarios is denoted by $S$.

The SCFLP is formulated with variables indicating which facilities are opened and what facility serves each of the customers. If facility $i \in I$ is open, the variables $x_i$ equal 1, 0 otherwise; opening facility $i$ incurs a cost of $f_i$. The variable $y^s_{ij}$ equals 1 to indicate that customer $j \in J$ is serviced by facility $i \in I$ in scenario $s \in S$, which has a cost of $q_{ij}$. Using these variables, the formulation of the SCFLP is given by

$$\min \quad \sum_{i \in I} f_i x_i + \frac{1}{|S|} \sum_{s \in S} \sum_{i \in I} \sum_{j \in J} q_{ij} y_{ij}^s, \tag{24}$$

$$\text{s.t.} \quad \sum_{i \in I} y_{ij}^s \geq \lambda_j^k \qquad \text{for all } j \in K, s \in S, \tag{25}$$

$$\sum_{j \in J} y_{ij}^s \leq k_i x_i \qquad \text{for all } i \in I, s \in S, \tag{26}$$

$$\sum_{i \in I} k_i x_i \geq \max_{s \in S} \sum_{j \in J} \lambda_j^s, \tag{27}$$

$$x_i \in \{0, 1\} \qquad \text{for all } i \in I, \tag{28}$$

$$y_{ij}^s \geq 0 \qquad \text{for all } i \in I, j \in J, s \in S. \tag{29}$$

The demand of customer $j$ in scenario $s$ is denoted by $\lambda_j^s$ and the capacity of facility $i$ is denoted by $k_i$. The constraints (25) ensure that the demand for each customer is satisfied in every scenario. Constraints (26) limit the customers served by each facility to the respective capacities. Finally, constraints (27) ensure that enough facilities are opened to cover the demand of all customers. This final constraint is redundant, but is useful in ensuring feasibility of the first stage decisions in the decomposed problem.

The instances included in this example have been collected from the OR-Library [6]. Within the reader (`reader_cap`), a parameter is provided to select the number of scenarios. These scenarios describe the demand for each customer $j$, which is sampled from a normal distribution with a mean of $\mu_j$, which is the deterministic demand given in the instance file, and a standard deviation sampled from a uniform distribution in the range $[0.1\mu_j, 0.3\mu_j]$.

By default, SCFLP instances are constructed as the monolithic deterministic equivalent (24)–(29) and solved directly by SCIP. The parameter `reading/cap/usebenders` can be set to `TRUE` to decompose the SCFLP using Benders' decomposition. The master problem consists of the $x_i$ variables and one subproblem is constructed for each scenario, comprising variables $y_{ij}^s$. The default Benders' decomposition plugin is used to interact with the framework provided in SCIP 6.0. Since the decomposition of SCFLP results in continuous subproblems, the classical Benders' optimality and feasibility cuts are generated for the instances. The cut generation methods are provided by `benderscut_opt` and `benderscut_feas`.

### 4.3 SCIP-Jack: Steiner Tree and Related Problems

Given an undirected, connected graph $G = (V, E)$, costs (or weights) $c : E \to \mathbb{R}_+$ and a set $T \subseteq V$ of *terminals*, the *Steiner tree problem in graphs* (SPG) asks for a tree $S = (V(S), E(S)) \subseteq G$ such that $T \subseteq V(S)$ holds and $\sum_{e \in E(S)} c(e)$ is minimized. The SPG is one of the fundamental combinatorial optimization problems [34] and the subject of more than a thousand research articles. Moreover, many related problems have been extensively described in the literature and can be found in a wide range of practical applications [23].

The SCIP Optimization Suite contains SCIP-Jack, an exact solver not only for the SPG, but also for 11 related problems. This release of the SCIP Optimization Suite contains the new SCIP-Jack 1.3. Most changes in the latest release concern the SPG, the *maximum-weight connected subgraph problem* (MWCSP), and the *prize-collecting Steiner tree problem* (PCSTP). Improvements for the SPG include a tentative implementation of some extended reduction techniques described by Polzin [52] and a new propagation routine. For the MWCSP, the reductions by Rehfeldt and Koch [53] have been implemented. Moreover, in the dual-ascent routine [55] a specialized extension for MWCSP and PCSTP has been added that improves the run time of this routine by a factor of more than 2 for most instances. Finally, for the PCSTP bottlenecks in the graph data structures and the ancestor data structures have been removed.

The developments sketched above yield the following results (with CPLEX 12.7.1 as LP solver). Several large-scale PCSTP instances (from the *HAND* test set) can now be solved more than two orders of magnitude faster than with SCIP-Jack 1.2. Furthermore, most MWCSP instances tested in [54] can now be solved more than twice as fast. The previously open benchmark instance cc7-3nu from the 11th DIMACS Challenge (which was originally formulated as a PCSTP, but can be transformed to MWCSP due to its single-weight edges) can be solved by SCIP-Jack 1.3 for the first time to optimality. Moreover, SCIP-Jack 1.3 participated in the *Parameterized Algorithms and Computational Experiments* (PACE) Challenge 2018 [49], dedicated to the Steiner tree problem in graphs, which allows for fixed-parameter algorithms in the number of terminals, and in the treewidth. Although SCIP-Jack does not implement any fixed-parameter algorithms, it finished 3rd place in Track A (exact solution of problems with few terminals), 1st place in Track B (exact solution of problems with bounded treewidth), and 2nd in Track C (heuristic solution of problems with different structures). In the PACE Challenge SoPlex 4.0 was used as the LP solver.

The focus of the future development will be on the SPG and the PCSTP. For both problems additional reduction techniques will be implemented, existing ones extended, and domain propagation will be improved. Some of the data structures for the PCSTP will be reimplemented, as they are still a bottleneck for some large-scale instances. Future work will also concentrate on the generation of new cutting planes. Finally, the dual-ascent routine will be improved, both algorithmically and implementation-wise. As dual-ascent is used for all problem classes covered by SCIP-Jack, any improvements of this routine have an overall impact on the performance of the solver.

## 5 The GCG Solver

GCG turns SCIP's branch-price-and-cut (BP&C) framework into a *generic branch-price-and-cut solver*. It performs a Dantzig-Wolfe reformulation [15] of the ("original") input MIP and solves the reformulated ("master") model with BP&C. That is, the relaxation in each node of the branch-and-bound tree is solved by column generation. The pricing subproblems are usually MIPs themselves and solved as sub-SCIPs or specialized solvers. It has always been GCG's ambition to not require the user to have any knowledge about problem or model structure or the decomposition algorithms to exploit such structure. An aim is to make decomposition methods more widely applicable also to non-experts.

A critical step is the automatic detection of such model structure that allows for a beneficial Dantzig-Wolfe reformulation. The new release GCG 3.0 incorporates major improvements in this respect. Moreover, thanks to the generic Benders' decomposition framework that was added to SCIP 6.0 (see Section 2.1), GCG has become a generic Benders' decomposition solver for MIPs. The new detection is flexible enough to support this case equally well. Again, the user does not need to provide or even know any model structure. In summary, this makes GCG 3.0 a general-purpose decomposition solver.

Preliminary results indicate that GCG 3.0 is faster than the previous version GCG 2.1.4 on structured instances and fails less often on unstructured ones.

## 5.1 A Modular Detection Loop

A model structure that can be exploited by a decomposition algorithm can be explicitly given, for example in form of a `dec` file. Otherwise, a suitable structure in the constraint matrix of the MIP must be identified. This structure itself is referred to as a *decomposition*, which can be exposed by suitably permuting rows and columns. An example on the MIPLIB 2010 instance `b2c1s1` can be seen in Figure 3. The classic single-bordered block-angular form for Dantzig-Wolfe is a set of independent subsystems of variables and constraints (the "blocks") which are linked by constraints that contain variables from more than one block. Linking variables that appear in more than one block can be present, and in such cases GCG applies a Lagrangean decomposition.

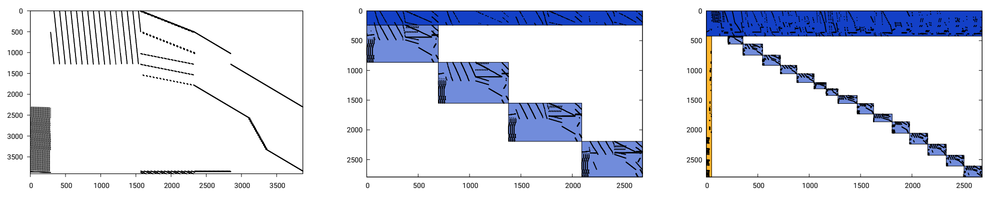

**Figure 3:** Nonzero structure of the constraint matrix of `b2c1s1`: original as given in the file, rearranged into a single-bordered block-diagonal form with only linking constraints and 4 blocks, and rearranged into a double-bordered block-diagonal form with linking constraints, linking variables, and 15 blocks. GCG 3.0 default detects 262 further decompositions of the presolved `b2c1s1` instance.

The previous versions of GCG detected model structure in a rather static way. Several *detectors* each worked on the entire matrix and, when successful, output one or more decompositions. An important class of detectors are graph-based detectors [8, 64]. The constraint matrix is, e.g., represented as a hypergraph $H$: for each nonzero entry there is a vertex and each row (and/or column) is represented by a hyperedge. A partition of $H$ into $k$ connected components reveals $k$ blocks in the matrix. GCG uses HMETIS to heuristically solve the hypergraph partitioning problem.

Our influence on the detected decompositions from this approach is limited by the parameters available in HMETIS. Since the hypergraph is partitioned only heuristically, very regular textbook decompositions of structured models, see Figure 4, were not consistently detected.

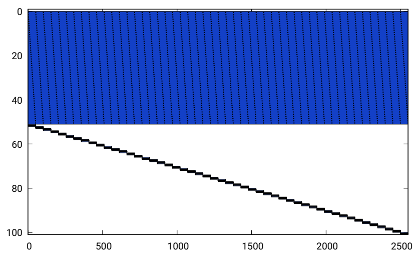

**Figure 4:** Nonzero structure of the constraint matrix of the capacitated $p$-median instance `p2050-1` with 50 blocks. A hypergraph-based approach would need to remove as many as halve of the hyperedges to find 50 connected components.

### 5.1.1 Orchestrating Detectors for Partial Decompositions

The mentioned shortcomings motivated a complete redesign of the structure detection in GCG 3.0. Each detector now works on one atomic concept at a time and fixes only a part of the decomposition, for example, the constraints of the master problem or a subset of it. That is, each detector receives a *partial decomposition* as input, in the code referred to as `seeed`,[7] where the roles of only a subset of constraints and variables are already given, initially none at all. Then the detector fixes the roles of additional constraints and/or variables, not necessarily all of them, and outputs a set of partial and/or complete decompositions. There are `detection/maxrounds` consecutive rounds in which detectors are called, iteratively refining partial decompositions. Detection is performed in parallel on partial decompositions. Decisions taken in previous rounds cannot be revoked, except by so-called postprocessors, see below. Different ways may lead to the same decomposition, but duplicates are filtered. Thus, we may represent the set of partial decompositions as a tree with complete decompositions at the leaf nodes. An example is given in Figure 5.

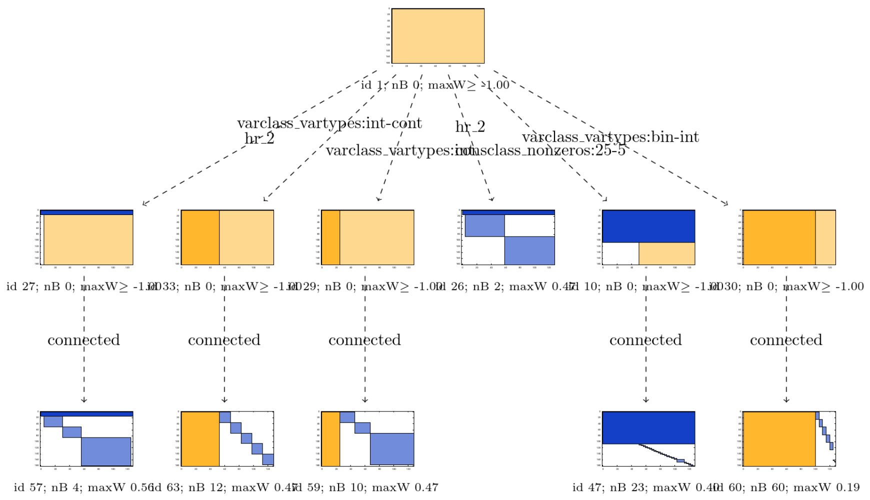

**Figure 5:** Example of a family tree of decompositions for `noswot`.

Each detector implements up to three callbacks to *propagate*, *finish*, or *postprocess* a (partial) decomposition. The first two work on partial decompositions. Propagators may output a mix of partial and complete decompositions while finishers always complete a decomposition. A typical finisher looks for connected components in the graph that represents only the yet unfixed constraints and variables of the matrix. A postprocessor may modify a complete decomposition, e.g., by re-assigning constraints from the master to a block, if applicable. Whether a specific detector is called in a specific round depends on the detection history, that is, whether certain detectors (maybe itself) were run/successful in previous rounds.

The modular detection scheme allows for finding structures that were undetected or even undetectable before GCG 3.0. One example is an uncommon structure like a *double-bordered staircase form*. Staircase structures appear in multi-stage optimization problems like lot-sizing. The corresponding detectors are enabled by setting the parameters `detection/detectors/stair*/enabled` to `TRUE`. When no decomposition is detected, GCG runs in “SCIP-mode” and solves the model with empty master problem and one block.

[7] Friends of German dancehall may recognize this as a reference to the esteemed Berlin-based band.

A `detection/emphasis` meta parameter has been included in the new release. The `default` uses only one round of only a few detectors, but still rather quickly finds a lot of classical decompositions. If too slow, the emphasis `fast` can be tried. Rather for experimental purposes, the emphasis can be set to `aggressive`, which enables all detectors and increases round limits. This emphasis can easily prove too expensive. If one is looking for a certain decomposition, the user is advised to enable or disable detectors directly and change their parameters accordingly, such as the number of blocks to look for.

### 5.1.2 Constraint and Variable Classifier

We assume that MIP modelers typically define *sets* of constraints, for capacities, assignments, implications, flow conservation, supplies, etc. In a classical Dantzig-Wolfe reformulation, each such set either distributes among blocks, often in a regular fashion, or completely remains in the master problem, see again Figure 4. This knowledge has not been systematically exploited in the past. It motivates the following strategy. Constraints are grouped into classes and entire classes are tentatively assigned to the master problem, resulting in a potentially large number of partial decompositions. Examples for classification criteria for constraints are the number of nonzeros, constraint type according to SCIP's constraint handlers or the classification used by MIPLIB 2010 [36], or constraint names that differ only by small edit distance. Typically, too many classes are identified to test all subsets, so that their number is limited to `detection/maxnclassesperclassifier`. The same modeling rationale motivates classifying variables; entire classes are tentatively assigned to become linking or static variables, which appear exclusively in the master problem. Variables are classified according to their SCIP type, (the sign of) their objective function coefficient, etc. The classification is performed before entering the detection loop and subsets of classes are assigned by propagating detectors `consclass`, and `varclass`.

Note that the classification according to similarity of constraint names gives a convenient way of annotating a model such that a particular decomposition is detected.

### 5.1.3 Interaction with Preprocessing

Preprocessing and symmetry handling can significantly modify the constraint matrix by deletion and addition of rows, columns, and nonzero coefficients. Since preprocessors are currently not "decomposition aware", a decomposable structure may not be recognized as such by the detectors after preprocessing has been applied. The balance between preprocessing and detection has not yet been addressed sufficiently in the literature. In GCG, emphasis is given on structure detection. Therefore, conflict analysis and several of SCIP's presolvers are not included. With the aim of recovering some information lost in preprocessing, GCG optionally performs constraint/variable classification in the original problem and may also run a full detection on the original problem (`detection/origprob/{classificationenabled,enabled}`). The resulting information is used in the detection on the preprocessed problem: Classes and decompositions are tried to be matched with/translated to those from the preprocessed model.

### 5.1.4 Guessing Candidates for the Number of Blocks

The hypergraph partitioning detectors `h{r,c,rc}gpartition` need as input the number of connected components (blocks) to look for. Wang and Ralphs [64] suggested to count the number of constraints with identical number of nonzeros, letting the constraints "vote" for numbers of nonzeros. A number of votes that appears most often

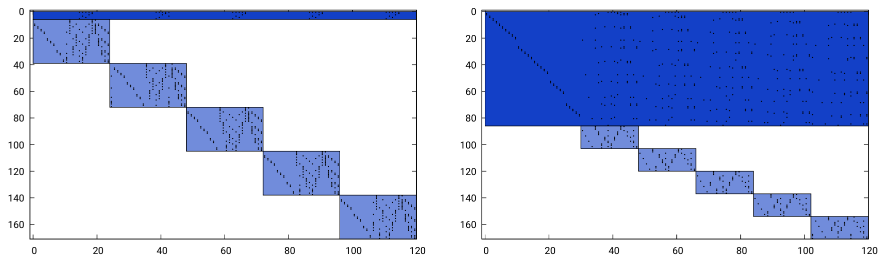
**Figure 6:** Two decompositions of `noswot` with 5 blocks each, but different numbers of master constraints and static variables; the white scores are 0.7719 and 0.4225, respectively.

is considered a candidate for the number of blocks. This implicitly assumes that all constraints of a certain type are distributed to the blocks. In contrast, GCG collects votes for greatest common divisors of cardinalities of constraint and variable classes. The `detection/detectors/h*partition/maxnblockcandidates` most likely candidates are then used by the hypergraph-based detectors. This list can be manually extended by setting the `detection/addblocknr` parameter.

### 5.1.5 Choosing a Final Decomposition to Work With

Different decompositions usually lead to vastly different computation times for solving the Dantzig-Wolfe reformulation. Experiments suggest that for MIPLIB models fewer constraints in the master problem may help [8], but further experimentation is necessary. GCG explicitly rewards a "textbook decomposition" like a set partitioning master problem with the remainder decomposing into (identical) blocks. There are other predefined `detection/scoretypes` that reflect modeling experience from the column generation literature. Experimentally, these features are weighted and combined with the percentage of the "white area" of a decomposition, illustrated in Figure 6. The highest score determines the final decomposition.

### 5.1.6 Updates to the `dec` File Format

The `dec` file format describes decompositions. A `dec` file always refers to variable and constraint names of a corresponding MIP. The format is section oriented where keywords in a separate line start a new section. Keywords are followed by one or more values, each in a separate line. Examples are `mastercons`, followed by names of constraints that are assigned to the master problem; and `presolved`, followed by `1` or `0` to indicate whether the decomposition refers to a presolved model or not.

In previous versions of the format, constraints that were not listed were assigned to the master. For backwards compatibility, this is still the default, but to make use of the new concept of partial decompositions, keyword `consdefaultmaster` followed by a `0` enables that constraints not listed are left unassigned. Moreover, linking and static variables can now be specified. Note that there are several implicit assignments of constraints or variables. For instance, constraints with variables that appear in several blocks are assigned to the master. Further details on the format can be found in the documentation of the file `reader_dec.h`.

### 5.1.7 Computational Results

The most noticeable improvement over previous versions is the robustness of structure detection. The new loop consistently finds decomposable structure, not only in "structured" models for which we know that they are amenable to a Dantzig-Wolfe reformulation, but also for "non-structured" models from the MIPLIBs. Figure 7 gives some results for all 361 instances of the MIPLIB 2010. Even the very basic setting of using constraint classifiers and the `connected` finisher only fails to detect a decomposition 10 times (among which 3 are timeouts). The upcoming release of the next MIPLIB will contain structure information provided with GCG 3.0.

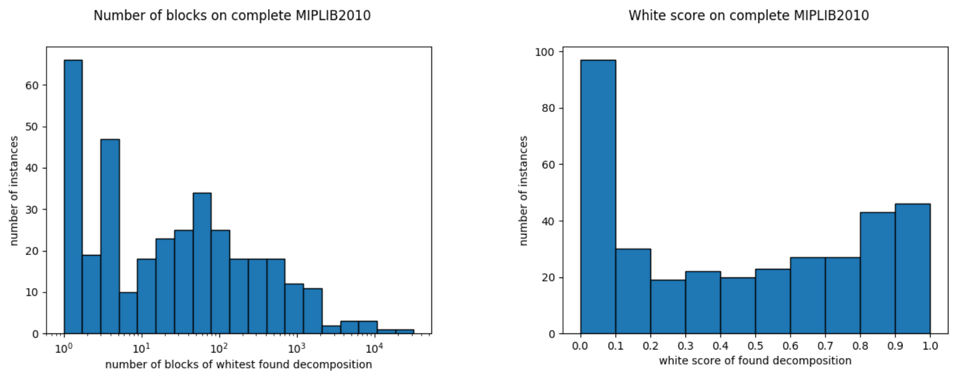

**Figure 7:** For almost 300 out of 361 instances of the entire MIPLIB 2010 we detect decompositions with at least 2 blocks; half of the instances have decompositions with more than 20 blocks (left figure). The score distribution of the respective "whitest" detected decomposition is given on the right.

## 5.2 Automatic Benders' Decomposition

As described in Section 2.1, SCIP 6.0 provides new Benders' decomposition functionality. GCG 3.0 can be seen as one frontend to this functionality, see Section 2.1.1. It automatically detects a Benders' decomposable structure as seen, for instance, in Figure 8. Then it forms a master problem with a series of subproblems and passes it to SCIP's Benders' decomposition framework to manage the subproblem solving and Benders' cut generation. The Benders' decomposition mode is activated by setting `relaxing/gcg/mode` to `1`.

Three settings are provided to detect structures suitable for Benders' decomposition: `detect-benders`, `detect-benders-bin_master`, and `detect-benders-cont_subpr`. The last two restrict detection to the special cases of having only binary variables in the master problem, or having only continuous variables in the subproblems, respectively. The last case comes closest to the structure to which CPLEX, since version 12.8, can apply Benders' decomposition. All settings use an experimental Benders' score to evaluate Benders' decompositions. In contrast to detecting Dantzig-Wolfe reformulations, the score encourages linking variables and heavily penalizes linking constraints. In particular, linking constraints must not share variables with the blocks. In a first computational comparison of GCG 3.0 against CPLEX 12.8 with Benders `auto-decompose model` mode on, CPLEX is clearly faster on MIPLIB models that both can solve to optimality. However, GCG is much more successful in identifying structure amenable to Benders' decomposition.

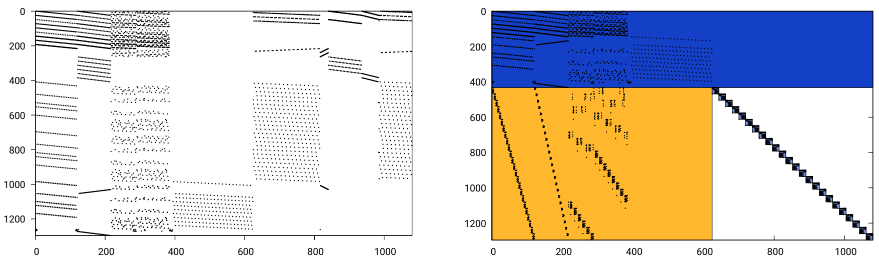

**Figure 8:** Nonzero structure of the constraint matrix of `gesa3`; original and rearranged into a form amenable to Benders' decomposition, as given in (4)–(5).

## 5.3 More Flexible Pricing

Pricing problems in GCG are usually MIPs and a sub-SCIP is used as the so-called *pricing solver*. This refers to the `pricingsolver/mip`. CPLEX is used when GCG is built with option `PRICINGSOLVER=cplex`. Beyond using a general MIP solver, there is also the possibility to apply specialized solvers. When the subproblem is detected to be a knapsack problem, it is solved with the solver specified in `pricingsolver/knapsack`. A new such pricing solver was added in GCG 3.0 for *weighted independent set* pricing problems (like for vertex coloring models). It uses the CLIQUER library and needs compilation with `CLIQUER=true`.

More importantly, the pricing scheme has been completely restructured. This in particular affects the way in which *heuristic pricing* is performed, and in which order the pricing solvers are applied on the pricing problems. Previously, exact pricing was not performed until all pricing problems had been solved heuristically, or heuristic pricing had been aborted due to a parameter limit. Furthermore, the pricing loop iterated over all pricing problems, while the next pricing problem was not considered until all available solvers had been tried on each pricing problem.

GCG 3.0 offers a more flexible pricing scheme. Pricing is now performed via *pricing jobs*, which essentially consist of a pricing problem, a solver to be applied on it, and the information whether the solver should be applied heuristically. There is a one-to-one correspondence between pricing jobs and pairs of pricing problems and solvers. These pricing jobs are organized in a priority queue, which is maintained by a new class, the *pricing controller*. The latter is also responsible for deciding over premature abortion of the loop. For example, the loop can be aborted if a certain percentage of the pricing problems have found at least one improving column, or if a certain number of columns has already been generated.

In this context, the way in which heuristic pricing is used has changed. During the pricing loop, the MIP and CPLEX pricing can now be applied more than one heuristic pricing iteration per pricing problem, i.e., if a (node, gap, solution) limit is reached, another iteration with an increased working limit may be performed rather than solving the problem to optimality.

## 5.4 More Statistics and Exploration Capabilities

After detection has finished, the `explore` menu contains the list of all detected decompositions with information about the detectors that found the decompositions, their score, etc. The default typically finds a few dozens up to a few hundred. The user can interactively `visualize` single decompositions in the PDF viewer `visual/pdfreader`, which defaults to EVINCE, or `select` them for further processing: to be used for Dantzig-Wolfe or Benders' reformulation, or in reporting. The PDF generation requires GNUPLOT. All figures in this report were produced using this functionality.

Furthermore, the `write` menu contains two new options to write LaTeX files, `familytree`, and `reportdecompositions`. The former produces the tree of (partial) decompositions as mentioned above, see Figure 5. The latter gives a detailed report, with figures, about all the decompositions found. The selection can be limited with the `select` command in the `explore` menu.

### 5.5 Further Improvements

Many parts of the code were revisited and improved, among others:

– The column management is now similar to SCIP's cut management with pricerstore and column pool.

– The dual variable stabilization is a hybridization of smoothing and an ascent method of Pessoa et al. [51]. It is enabled in the root node only.

– The LP relaxation of the original model is now solved for obtaining a first dual bound and to run SCIP's heuristics.

– The initialization of the master problem also benefits from this and from the introduction of artificial variables combined with a "big $M$" method, when this is numerically safe.

– Discretization is now enabled on MIPs where continuous variables are convexified. This allows for aggregation of subproblems also in the mixed-integer case.

– Several primal heuristics were improved. GCG can now be used as the popular price-and-branch heuristic by setting `heuristics/restmaster/pbheur` in the `master` menu to `TRUE`.

## 6 The UG Framework

The *Ubiquity Generator* (UG) is a generic framework for parallelizing branch-and-bound solvers. The SCIP Optimization Suite contains UG parallelizations of SCIP for both shared and distributed memory computing environments, namely FiberSCIP [61] and ParaSCIP [60]. A more detailed recent overview of the UG framework is given by Shinano [59]. The release UG 0.8.5 extends the framework by

– the availability of *`C++11` threads* for shared memory communication,

– *customized racing ramp-up*, which allows to use a set of user-defined parameters for the racing ramp-up phase,

– branching on constraints for ug[SCIP,*], and

– user interfaces to initialize subproblems for ug[SCIP,*].

Previously, Pthreads was the only communicator available for shared memory computing environments, but with the latest version of UG also *`C++11` threads* can be used. Since `C++11` threads are a programming language feature, their usage improves the portability of the parallel solver instantiated by UG.

Table 4 presents a brief performance comparison between FiberSCIP with Pthreads and with `C++11` threads on the 87 instances of the MIPLIB 2010 benchmark set. For the experiment a PC cluster with Intel Xeon E5-2670 v2 CPUs with 2.50GHz, two sockets with 10 cores each, and with 125 GB of main memory was used. Both FiberSCIP versions are agorithmically equivalent and use distributed domain propagation [27] during racing ramp-up. Since UG's parallelization is not deterministic, the results are aeraged over 5 repeated runs. As the comparison shows, the performance of FiberSCIP with

**Table 4:** Comparison between SCIP with Pthreads and `C++11` threads as communicator, using 4 and 12 threads. The performance of sequential SCIP is given as a reference. Statistics reported are the geometric mean of solving time in seconds and the number of solved instances/timeouts/aborts.

| Comm. | None | | Pthreads | | | | `C++11` | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Run | SCIP | | FiberSCIP (4th) | | FiberSCIP (12th) | | FiberSCIP (4th) | | FiberSCIP (12th) | |
| 1 | 446.5 | (75/12/0) | 366.1 | (77/10/0) | 299.3 | (79/7/1) | 351.1 | (78/9/0) | 302.3 | 78/8/1 |
| 2 | | | 360.1 | (77/9/1) | 292.1 | (79/8/0) | 371.9 | (77/10/0) | 285.5 | 81/6/0 |
| 3 | | | 371.3 | (76/11/0) | 291.1 | (78/9/0) | 361.6 | (78/9/0) | 291.5 | 81/5/1 |
| 4 | | | 352.5 | (77/10/0) | 301.2 | (78/8/1) | 359.6 | (78/9/0) | 296.3 | 78/8/1 |
| 5 | | | 354.1 | (77/10/0) | 304.7 | (79/6/2) | 362.7 | (77/10/0) | 284.6 | 80/7/0 |
| Avg. | 446.5 | | 360.8 | | 297.7 | | 361.4 | | 292.0 | |

Pthreads and `C++11` threads is quite similar, but with `C++11` threads FiberSCIP seems to be slightly faster when using 12 threads. Therefore, FiberSCIP with `C++11` threads has become the new default in this release.

Note that the parameters are not tuned for the latest release version of SCIP and the testset from MIPLIB 2010 benchmark is not overly suited for achieving large performance gains by parallelization. The reason is that by now many of the instances are either easy to solve with a single thread or too hard for SCIP even using multiple threads in a few hours. Nevertheless, it can be observed that FiberSCIP outperforms the sequential version of SCIP performance in parallelization on the PC up to 12 threads. A closer analysis shows that on average more than 43 instances were solved in racing stage when using 4 threads and more than 45 instances with 12 threads. Hence, for this testset, the *racing mechanism* seems to have a large performance impact than the actual parallelization of the search tree.

Racing ramp-up is a UG feature to exploit the performance variability commonly observed in MIP solving. An instance is solved multiple times in parallel, each time with a different parameter setting. If the instance has not been solved to optimality when a predefined termination criterion, e.g., a time limit, is reached, the most promising branch-and-bound tree is distributed among the UG solvers and the default solving procedure is initiated. The latest UG release includes *customized racing*, which allows the user to specify their own parameter settings for racing. If the number of UG solvers exceeds the number of provided parameter sets, then the customized parameter settings are combined with default ones. While this release version of SCIP does not use customized racing by default, it is applied in both ug[SCIP-Jack,*] and ug[SCIP-SDP,*].

This release also contains several updates regarding the parallel version of the Steiner tree problem solver SCIP-Jack (see Section 4.3). Instead of branching on variables, which in the case of Steiner tree problem correspond to edges, default SCIP-Jack uses vertex branching [30]. During the branch-and-bound process, SCIP-Jack selects a non-terminal vertex of the Steiner problem graph to be rendered a terminal in one branch-and-bound child node and to be excluded in the other child. These two operations are modeled in the underlying IP formulation by including one additional constraint. While this procedure could not be used in previous versions of ug[SCIP-Jack,*], the latest UG release now allows to not only branch on variables, but also on constraints. Furthermore, to retain previous branching decisions, ug[SCIP-Jack,*] transfers the branching history together with a subproblem, enabling SCIP-Jack to change the underlying graph (adds terminals and deletes vertices). Additionally, whenever a subproblem has been transfered, SCIP-Jack performs aggressive reduction routines to reduce the problem further.

For ug[SCIP-SDP,*] the customized racing can be used to combine both nonlinear branch-and-bound and an LP-based cutting plane approach into a single solver for mixed-integer semidefinite programs. In this case, one or possibly more threads will try to solve the root node and the earlier parts of the tree using linear relaxations and eigenvector cuts. The remaining threads will solve semidefinite relaxations until after the racing stage the more promising approach on the particular instance is chosen for the remainder of the solution process. This allows to exploit the sometimes very fast but on other types of problems almost non-progressing cutting plane approach whenever it works well on a particular instance, while keeping the more stable behavior of the SDP-based branch-and-bound approach. Some first results on the CBLIB [19] show that this approach leads to a speedup factor of three on the cardinality-constrained least-squares instances already for two threads because of the success of the cutting-plane approach on these instances. The LP-based approach also helps on some of the truss topology instances, which also benefit from the parallelization of the branch-and-bound tree, while the partitioning instances with smaller trees do not lend themselves well to this kind of parallelization, since they are also not really suited for the LP-approach.

## 7 Final Remarks

The most notable extensions of functionality in Release 6.0 of the SCIP Optimization Suite has been the support for Benders' decomposition, both in SCIP and GCG. This presents a first step towards the tighter integration between the decomposition algorithm and state-of-the-art solvers for mixed-integer optimization. From a research perspective, a generic Benders' decomposition framework of this form allows to measure and compare the impact of algorithmic ideas beyond single problem-specific implementations. With the interaction with large neighborhood search heuristics one showcase for this advantage has already been displayed. Future research will extend the framework with the addition of enhancement techniques and new cutting plane methods and investigate methods to detect structures most amenable to the application of Benders' decomposition.

Further, through GCG, the Benders' decomposition framework makes SCIP one of the first state-of-the-art solvers capable of automatically applying Benders' decomposition to general MIP instances, also for non-expert users. This added usability was made possible first and foremost through a major redesign and many improvements in the core of GCG, that are equally targeted towards its classical use case for performing Dantzig-Wolfe reformulation generically and automatically. The greatly increased flexibility of the new modular detection scheme allows for finding structures that were undetected or even undetectable with previous versions.

Last, not least, it should be mentioned that SCIP's MINLP performance was improved by careful tuning and the performance of the MIP core was advanced through two algorithmic innovations: a new diving heuristic called Farkas diving, which exploits interactions with conflict analysis, and a new cut selection measure called the directed cutoff distance, which takes into account knowledge about the incumbent solution. All in all, these enhancements help to reduce the average running time of SCIP and increase the number of solved instances for both MIP and MINLP testsets.

**Acknowledgements**

The authors want to thank all contributors to the SCIP Optimization Suite and all users that reported bugs. For this release, special thanks go to Tobias Achterberg and Michael Winkler for supporting the LP interface to Gurobi, to Stefan Heinz for improving the LP interface to Xpress, and to Anass Meskini and José Manuel García Salmerón for providing new implementations of MINLP models as part of the Callable Library example. Last, not least, many thanks go to Stefan Vigerske for his continuous support regarding the MINLP functionalities of SCIP.

## Code Contributions of the Authors

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

SCIP's extension for Benders' decomposition including the interface to PySCIPOpt and the stochastic capacitated facility location example (Section 2.1 and 4.2) have been implemented by SM. The nonlinear extensions of PySCIPOpt have been implemented by FeS. GH conducted the tuning of the ALNS heuristic and the new conflict-driven diving heuristics were contributed by JaW and AG (Section 2.3). The lookahead branching rule (Section 2.4) was added by CS and GG. RG implemented the improvements to the cutting plane separation (Section 2.5). The improvements to symmetry handling (Section 2.6) were performed by MP and CH. Several LPI interfaces and their unit tests have been maintained by MP with support from FrS, FeS, and AG (Section 2.7). The generalized variable locks have been implemented by JaW (Section 2.8.1). GH and FrS have supported the infrastructure for performing and evaluating computational experiments throughout the SCIP development with their work on the tools Ipet [31] and Rubberband [57].

The improvements to SoPlex described in Section 3 have been chiefly performed by MM together with GG (for the new aggregation presolver). The work on the applications in Section 4 has been conducted by BM, FW, and AG (for recursive ring packing), DR (updates in SCIP-Jack), and TG (updates in SCIP-SDP). The adjustments to UG explained in Section 6 have been implemented by YS, TG (for MISDP), and DR (for SCIP-Jack).

The redesign of GCG's structure detection described in Section 5.1 has been performed by MB, its interaction with the Benders' framework (Section 5.2) by SM. The visualization of (partial) decompositions has been implemented by MB. CP restructured the pricing loop as described in Section 5.3 and improved several primal heuristics. JoW improved the initialization of the master problem and the implementations of column and cut management and the stabilization of dual variables. Build files for cmake were added by MW.

## 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 R. Wunderling. Mixed integer programming: Analyzing 12 years of progress. In M. Jünger and G. Reinelt, editors, *Facets of Combinatorial Optimization: Festschrift for Martin Grötschel*, pages 449–481. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-38189-8_18.

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

[5] G. Andreello, A. Caprara, and M. Fischetti. Embedding $\{0, \frac{1}{2}\}$-cuts in a branch-and-cut framework: A computational study. *INFORMS Journal on Computing*, 19(2):229–238, 2007. doi:10.1287/ijoc.1050.0162.

[6] J. E. Beasley. Or-library: Distributing test problems by electronic mail. *The Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

[7] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4(1):238–252, 1962. doi:10.1007/BF01386316.

[8] M. Bergner, A. Caprara, A. Ceselli, F. Furini, M. Lübbecke, E. Malaguti, and E. Traversi. Automatic Dantzig-Wolfe reformulation of mixed integer programs. *Mathematical Programming*, 149(1–2):391–424, 2015. doi:10.1007/s10107-014-0761-5.

[9] T. Berthold. Heuristics of the Branch-Cut-and-Price-Framework SCIP. In J. Kalcsics and S. Nickel, editors, *Operations Research Proceedings 2007*, pages 31–36, 2008.

[10] J. R. Birge, M. A. Dempster, H. I. Gassmann, E. Gunn, A. J. King, and S. W. Wallace. A standard input format for multiperiod stochastic linear programs. Technical Report WP-87-118, IIASA, Laxenburg, Austria, 1987.

[11] N. Boland, M. Fischetti, M. Monaci, and M. Savelsbergh. Proximity Benders: a decomposition heuristic for stochastic programs. *Journal of Heuristics*, 22(2):181–198, 2016. doi:10.1007/s10732-015-9306-1.

[12] C. Carøe and J. Tind. L-shaped decomposition of two-stage stochastic programs with integer recourse. *Mathematical Programming*, 83(1):451–464, 1998.

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

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

[15] G. B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8(1):101–111, 1960. doi:10.2307/167547.

[16] S. S. Dey and M. Molinaro. Theoretical challenges towards cutting-plane selection. *Mathematical Programming*, 170(1):237–266, 2018. doi:10.1007/s10107-018-1302-4.

[17] L. Eifler. Mixed-integer programming for clustering in non-reversible Markov processes. Master's thesis, Technische Universität Berlin, 2017.

[18] A. A. Farley. A note on bounding a class of linear programming problems, including cutting stock problems. *Operations Research*, 38(5):922–923, 1990. doi:10.1287/opre.38.5.922.

[19] H. A. Friberg. CBLIB 2014: A benchmark library for conic mixed-integer and continuous optimization. *Mathematical Programming Computation*, 8(2):191–214, 2016.

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

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

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

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

[24] W. Glankwamdee and J. Linderoth. Lookahead branching for mixed integer programming. In *12th INFORMS Computing Society Conference (ICS2011)*, pages 130–147, 2011.

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

[26] A. Gleixner, S. Maher, B. Müller, and J. P. Pedroso. Exact methods for recursive circle packing. 2017. Under review.

[27] R. L. Gottwald, S. J. Maher, and Y. Shinano. Distributed domain propagation. Technical Report 16-71, ZIB, Takustr. 7, 14195 Berlin, 2016.

[28] C. Hojny and M. E. Pfetsch. Polytopes associated with symmetry handling. *Mathematical Programming*, 2018. doi:10.1007/s10107-018-1239-7.

[29] J. Hooker and G. Ottosson. Logic-based benders decomposition. *Mathematical Programming*, 96(1):33–60, 2003. ISSN 0025-5610.

[30] F. Hwang, D. Richards, and P. Winter. The Steiner tree problem. *Annals of Discrete Mathematics*, 53, 1992.

[31] Ipet. Interactive Performance Evaluation Tools for Optimization Software. `http://www.github.com/gregorch/ipet`.

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

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

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

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

[36] 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.

[37] G. Laporte and F. V. Louveaux. The integer L-shaped method for stochastic integer programs with complete recourse. *Operations Research Letters*, 13(3):133–142, 1993. doi:10.1016/0167-6377(93)90002-X.

[38] J. K. Lenstra and A. H. G. R. Kan. *Complexity of packing, covering and partitioning problems.* Econometric Institute, 1979.

[39] F. V. Louveaux. Discrete stochastic location models. *Annals of Operations Research*, 6 (2):21–34, 1986. doi:10.1007/BF02027380.

[40] T. Magnanti and R. Wong. Accelerating Benders' decomposition: algorithmic enhancement and model selection criteria. *Operations Research*, 29(3):464–484, 1981. ISSN 0030-364X.

[41] S. J. Maher. Large neighbourhood benders' search. Technical report, Optimization Online, 2018.

[42] S. J. Maher, T. Fischer, T. Gally, G. Gamrath, A. Gleixner, R. L. Gottwald, G. Hendel, T. Koch, M. E. Lübbecke, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, S. Schenker, R. Schwarz, F. Serrano, Y. Shinano, D. Weninger, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 4.0. Technical Report 17-12, ZIB, Takustr. 7, 14195 Berlin, 2017.

[43] H. Marchand and L. A. Wolsey. Aggregation and mixed integer rounding to solve MIPs. *Operations Research*, 49(3):363–371, 2001. doi:10.1287/opre.49.3.363.11211.

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

[45] D. McDaniel and M. Devine. A modified Benders' partitioning algorithm for mixed integer programming. *Management Science*, 24(3):312–319, 1977. doi:10.1287/mnsc.24.3.312.

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

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

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

[49] PACE. PACE 2018: Parameterized algorithms and computational experiments challenge. `https://pacechallenge.wordpress.com/pace-2018/`.

[50] J. P. Pedroso, S. Cunha, and J. N. Tavares. Recursive circle packing problems. *International Transactions in Operational Research*, 23(1-2):355–368, 2016. doi:10.1111/itor.12107.

[51] A. Pessoa, R. Sadykov, E. Uchoa, and F. Vanderbeck. In-out separation and column generation stabilization by dual price smoothing. In B. V., C. Demetrescu, and A. Marchetti-Spaccamela, editors, *Experimental Algorithms*, volume 7933 of *Lecture Notes in Computer Science*, pages 354–365, Berlin, 2013. Springer-Verlag.

[52] T. Polzin. *Algorithms for the Steiner problem in networks.* PhD thesis, Saarland University, 2004. URL `http://scidok.sulb.uni-saarland.de/volltexte/2004/218/index.html`.

[53] D. Rehfeldt and T. Koch. Generalized preprocessing techniques for Steiner tree and maximum-weight connected subgraph problems. Technical Report 17-57, ZIB, Takustr. 7, 14195 Berlin, 2017.

[54] D. Rehfeldt and T. Koch. Combining NP-Hard Reduction Techniques and Strong Heuristics in an Exact Algorithm for the Maximum-Weight Connected Subgraph Problem. Technical Report 17-45, ZIB, Takustr. 7, 14195 Berlin, 2017.

[55] D. Rehfeldt and T. Koch. Transformations for the prize-collecting steiner tree problem and the maximum-weight connected subgraph problem to SAP. *Journal of Computational Mathematics*, 36(3):459–468, 2018.

[56] W. Rei, J.-F. Cordeau, M. Gendreau, and P. Soriano. Accelerating Benders' decomposition by local branching. *INFORMS Journal on Computing*, 21(2):333–345, 2009. doi:10.1287/ijoc.1080.0296.

[57] Rubberband. A flexible archiving platform for optimization benchmarks. `http://www.github.com/ambros-gleixner/rubberband`.

[58] C. Schubert. Multi-Level Lookahead Branching. Master's thesis, Technische Universität Berlin, 2017.

[59] Y. Shinano. The Ubiquity Generator framework: 7 years of progress in parallelizing branch-and-bound. In N. Kliewer, J. F. Ehmke, and R. Borndörfer, editors, *Operations Research Proceedings 2017*, pages 143–149, Cham, 2018. Springer International Publishing.

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

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

[62] E. Thorsteinsson. Branch-and-check: A hybrid framework integrating mixed integer programming and constraint logic programming. In *Principles and Practice of Constraint Programming – CP 2001*, pages 16–30. Springer, 2001.

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

[64] J. Wang and T. Ralphs. Computational experience with hypergraph-based methods for automatic decomposition in discrete optimization. In C. Gomes and M. Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 394–402. Springer, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-38171-3_31.

[65] F. Wesselmann and U. H. Suhl. Implementing cutting plane management and selection techniques. Technical report, University of Paderborn, 2012.

[66] J. Witzig, I. Beckenbach, L. Eifler, K. Fackeldey, A. Gleixner, A. Grever, and M. Weber. Mixed-integer programming for cycle detection in non-reversible Markov processes. *Multiscale Modeling and Simulation*, 2016. Accepted for publication.

[67] J. Witzig, T. Berthold, and S. Heinz. Experiments with conflict analysis in mixed integer programming. In D. Salvagnin and M. Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming: 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings*, pages 211–220. Springer, Cham, 2017. doi:10.1007/978-3-319-59776-8_17.

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