Zuse Institute Berlin

Takustrasse 7
D-14195 Berlin-Dahlem
Germany

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

The SCIP Optimization Suite 4.0

ZIB Report 17-12 (revised version for release 4.0.1, September 2017)

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 4.0

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

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

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

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

[4]Friedrich-Alexander Universität Erlangen-Nürnberg, Cauerstr. 11, 91058 Erlangen, Germany, dieter.weninger@fau.de

revised version September 4, 2017, for release 4.0.1

Abstract. The SCIP Optimization Suite is a powerful collection of optimization software that consists of the branch-cut-and-price framework and mixed-integer programming solver SCIP, the linear programming solver SoPlex, the modeling language Zimpl, the parallelization framework UG, and the generic branch-cut-and-price solver GCG. Additionally, it features the extensions SCIP-Jack for solving Steiner tree problems, PolySCIP for solving multi-objective problems, and SCIP-SDP for solving mixed-integer semidefinite programs. The SCIP Optimization Suite has been continuously developed and has now reached version 4.0. The goal of this report is to present the recent changes to the collection. We not only describe the theoretical basis, but focus on implementation aspects and their computational consequences.

Keywords: linear programming, mixed-integer linear and nonlinear programming

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

1 Introduction

Mathematical programming, which includes mixed integer linear programming (MIP) and mixed integer nonlinear programming (MINLP), is a very valuable tool for many areas of academia and

industry. Its success stems from progress in both theory and practice, paralleled by the continuous development of software. Starting from the seminal contribution of Dantzig, Fulkerson, and Johnson [43] to computational mixed integer linear programming, software has become an important research tool for the development of new optimization methods. This has led to a long list of codes and software packages that have been developed to tackle problems of academic and industrial importance. An overview of these developments is presented by Bixby [22].

One culmination of the effort by research scientists and industry practitioners is the high quality software packages now available to solve MIP and MINLP problems. Some of the most prominent commercial solvers are CPLEX [81], Gurobi [77] and Xpress [51]. In addition, many successful non-commercial solvers have been developed such as Cbc [35], SYMPHONY [88], and SCIP (Solving Constraint Integer Programs) [2, 63].

A significant part of the success of many MIP and MINLP solvers comes from the keen attention to detail in the software implementation of fundamental theoretical ideas. Many of the most efficacious algorithms in MIP and MINLP solvers were derived from early theoretical contributions—particularly in relation to cutting planes [74, 43]. The effects from the implementation of key theoretical results are evident in the performance improvement of CPLEX [81] from version 6 to 6.5, see Bixby [22].

The constant progress in MIP and MINLP solver technology comes with a changing environment for software development. For many years, the repeated development of solvers to satisfy a singular purpose—the completion of a PhD thesis or to solve a particular industrial problem—was satisfactory. However, the growth of the field has led to many theoretical contributions and, equally important, technical advancements that have become necessary components of state-of-the-art MIP and MINLP solvers. This evolution has made it necessary to build future developments on flexible software frameworks.

This paper provides a snapshot of the development process for the mathematical programming solver SCIP. Many areas of the solver will be covered to provide the reader with a detailed understanding of the necessary technical and theoretical developments. This paper will deliver insights beyond the theoretical basis of the implemented algorithms by describing the fundamental technical considerations with the hope to inspire developers of mathematical programming solvers.

1.1 The SCIP Optimization Suite

From its conception as a Constraint Integer Programming (CIP) solver [2], SCIP [148] has evolved through the development of features, extensions and applications into an immensely versatile branch-cut-and-price framework and standalone CIP solver—with a particular focus on MIP and MINLP. Consider a *mixed integer nonlinear program* of the form

$$\begin{array}{lll} \min & f(x) & \\ \text{s.t.} & g_i(x) \leq 0 & \forall\, i \in \mathcal{M}, \\ & l_j \leq x_j \leq u_j & \forall\, j \in \mathcal{N}, \\ & x_j \in \mathbb{Z} & \forall\, j \in \mathcal{I}, \end{array} \tag{1}$$

where $\mathcal{I} \subseteq \mathcal{N} \coloneqq \{1, \ldots, n\}$ is the index set of integer variables, $f(x) \coloneqq \mathbb{R}^n \to \mathbb{R}$, $g_i \coloneqq \mathbb{R}^n \to \mathbb{R}$ for $i \in \mathcal{M} \coloneqq \{1, \ldots, m\}$ and $l \in (\mathbb{R} \cup \{-\infty\})^n$ and $u \in (\mathbb{R} \cup \{+\infty\})^n$ are lower and upper bounds on the variables respectively.

Different problem types within this class can be described by imposing restrictions on the constraints of (1). Specifically, SCIP can be employed to solve the following problems:

- *Mixed integer nonlinear program* (MINLP): given by (1).

- *Convex MINLP*: $f(x)$ and $g_i(x)$ are convex functions for all $i \in \mathcal{M}$.
- *Mixed integer quadratically constrained program* (MIQCP): $f(x)$ and $g_i(x)$ are quadratic functions for all $i \in \mathcal{M}$.
- *Mixed binary quadratic program* (MBQP): a MIQCP with the additional constraint that $x_j \in \{0, 1\}$ for all $j \in \mathcal{I}$.
- *Mixed integer program* (MIP): $f(x)$ and $g_i(x)$ are linear for all $i \in \mathcal{M}$.

In addition, if $\mathcal{I} = \emptyset$ then the above problem classes are called nonlinear program (NLP), convex NLP, quadratically constrained program (QCP), and linear program (LP), respectively. Furthermore, if several objectives $f_1(x), \ldots, f_k(x)$ for $k \geq 2$ are given, then the problem class is called *multi-objective program* (MOP). Throughout this paper, $f(x)$ is assumed to be linear. If $f(x)$ is nonlinear, an equivalent formulation with a linear objective function can be formed with one additional auxiliary variable and coupling constraint.

As its name suggests, SCIP allows to formulate and solve optimization problems including more general constraints in the sense of constraint programming. Such problems are described as *constraint integer programs* (CIP). The ability to solve CIPs has been achieved through the implementation of many effective and fundamental algorithms, such as constraint and domain propagation and conflict analysis.

Finally, an important feature of SCIP is the native support for branch-and-price. With the use of the pricer plugin type, users are able to employ branch-and-price without having to develop a tree search algorithm.

The continued development of the SCIP Optimization Suite has resulted in the introduction of many new, powerful features. This makes SCIP one of the fastest non-commercial MIP solvers, see Mittelmann [101]. The LP solver SoPlex [150] is a major component of the SCIP Optimization Suite and is fundamental for solving many of the problem classes handled by SCIP. A stalwart of the SCIP Optimization Suite is Zimpl [152], which is a very versatile mathematical programming modelling language. The collection of SCIP, SoPlex and Zimpl represent the founding software of the SCIP Optimization Suite.

Building upon the powerful branch-and-price framework of SCIP, GCG [146] provides user-driven or automatic Dantzig-Wolfe decomposition and supporting solution techniques like generic column generation. The value of GCG is the functionality that supports users in employing Dantzig-Wolfe decomposition, even as a non-expert. This includes state-of-the-art branching techniques, primal heuristics, and cutting planes. Additionally, GCG provides a platform for investigating decomposition techniques and related solution algorithms.

Parallelization has been important in pushing SCIP to solve previously unsolved instances from the MIPLIB 2003 [5] and MIPLIB 2010 [85] instance collections. The UG [151] framework was developed as an external parallelization framework for branch-and-bound solvers. Until now it has been predominantly used to parallelize SCIP, but this release features implementations for the base solvers PIPS-SBB [105] and Xpress [51]. Complementing the UG framework, the current release presents a first internal parallelization infrastructure for SCIP.

Many extensions and applications are provided within the SCIP Optimization Suite. Two very successful applications are the Steiner Tree Problem (STP) solver SCIP-Jack [65] and the multi-objective optimization solver PolySCIP [27, 147]. SCIP-Jack implements a state-of-the-art branch-and-cut based algorithm to solve 11 different Steiner tree problem variants. The current release introduces many new solving techniques and general enhancements that significantly improve the performance of SCIP-Jack. The multi-objective optimization solver PolySCIP enables users to compute non-dominated points for MOPs. This release includes

developments for PolySCIP that enable the computing all non-dominated points for integer problems with two and three objectives.

The plugin SCIP-SDP [149] further extends SCIP to solve semidefinite programs (SDPs) with integer variables. This new plugin can be combined with interior-point SDP solvers in a nonlinear branch-and-bound algorithm. Alternatively, SCIP-SDP can be used with a cutting plane approach, which is similar to how SCIP currently solves MINLPs. The current release provides an interface to the commercial SDP solver MOSEK [103]. Furthermore, the updated handling of relaxation solutions during enforcement, which was added for SCIP 4.0, allows SCIP-SDP to be combined with all constraint types implemented in SCIP to solve mixed-integer nonlinear semidefinite programs.

1.2 Structure of the paper

The features and contributions of the most recent release of the SCIP Optimization Suite will be sectioned by each component. Section 2 describes the technical and theoretical advancements that have been developed for SCIP. New interfaces for the SCIP Optimization Suite will be described in Section 3. The features developed for SoPlex, GCG, and UG will be presented in Sections 4, and 5. The updates to the SCIP-based applications of SCIP-Jack, PolySCIP and SCIP-SDP will be presented in Section 7. While the results from computational experiments will accompany each of the features, an overall performance analysis of SCIP in regards to MIP and MINLP will be presented in Section 8. Finally, Section 9 will conclude this report with some remarks on the presented improvements and developments of the SCIP Optimization Suite.

2 SCIP

The central pillar of the SCIP Optimization Suite is the CIP solver SCIP. This SCIP Optimization Suite release presents SCIP 4.0. Development effort has touched all aspects of the solver and has resulted in technical and theoretical contributions.

2.1 Technical improvements

As part of the ongoing development of SCIP, the improvement in the implementation of algorithms is important to maintain a strong solving performance. In this regard, much effort has been devoted to developing better algorithms for critical features of SCIP.

The current release presents technical improvements to the following features of SCIP:

- cycle detection in the variable bound graph (Section 2.1.1),
- connectivity of the conflict graph for clique partitioning (Section 2.1.2),
- the improved handling of connected components (Section 2.1.3)
- propagation loop and event handling system (Section 2.1.4),
- enforcement of relaxation solutions (Section 2.1.5),
- the ability to input partial solutions (Section 2.1.6)
- the use of randomization (Section 2.1.7),
- revised hash table implementation (Section 2.1.8).

2.1.1 Cycle detection in the variable bound graph

Given a linear constraint on two variables, for example $x + ay \leq b$ with $a \neq 0$, implicit bounds on each variable can be defined with respect to the alternate variable. In the given example these bounds would be $x \leq b - ay$ and $y \leq \frac{b}{a} - \frac{x}{a}$ (if $a > 0$, otherwise the relation symbol changes to $\geq$). Such bounds, which depend (affine linearly) on the value of exactly one other variable, are called *variable bound relations* or *v-bounds* in SCIP, see Achterberg [1]. V-bounds can be used among others to replace non-binary variables by binary ones when creating c-MIR cuts [97] or to guide primal heuristics [62].

During the presolving process of SCIP, v-bounds are extracted not only from linear constraints on two variables, but also from more general constraints and by probing [121]. The set of all v-bounds identified within a problem is stored in a global structure called the *variable bound graph*. This directed graph has two nodes per variable x_i, one for each of its bounds. These nodes are called $lb(x_i)$ and $ub(x_i)$, representing the lower and upper bound of x_i, respectively. Each arc in the graph represents a v-bound. For example, if $x_i \leq \alpha x_j + \beta$, the upper bound of x_i depends on the value of x_j. If x_j is not fixed yet and $\alpha > 0$ holds, $\alpha u_j + \beta$ forms a valid upper bound on x_i already, where u_j is the upper bound of x_j. This v-bound is represented by a directed arc pointing from $ub(x_j)$ to $ub(x_i)$. Analogously, if $\alpha < 0$ holds, $\alpha l_j + \beta$ establishes an upper bound on x_i, with l_j being the lower bound of x_j, and the variable bound graph would contain an arc pointing from $lb(x_j)$ to $ub(x_i)$. Figure 1 shows a set of constraints (a), the corresponding v-bound relations (b), and the resulting variable bound graph (c). In general, an arc in the variable bound graph represents the knowledge that a tightening of the bound represented

The variable bound graph is currently used by two algorithms implemented in SCIP: the variable bound propagator and the variable bound heuristic [62]. Both make use of an "almost topological ordering" of the nodes in the variable bound graph. Given an acyclic directed graph, a *topological ordering* of the nodes is an order of the nodes such that for each arc in the graph the tail precedes the head in that order. The variable bound graph in practical problems however, is not always acyclic, and so SCIP breaks cycles randomly (which gives an *almost* topological ordering). However, more information can be extracted from a cycle in the variable bound graph. SCIP 4.0 uses this additionally extracted information to deduce bound changes.

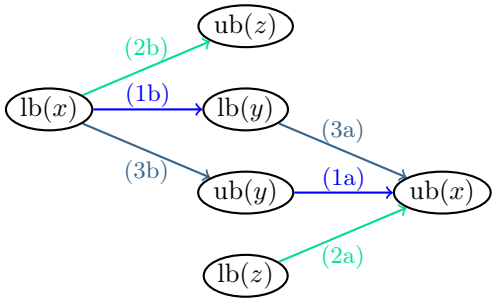
Feature description. A cycle in the variable bound graph means that the value of a given variable gives a bound for that same variable (after influencing the bounds of some variables in between). Essentially, this means that the cycle represents a v-bound of the form $x_i \leq \alpha x_i + \beta$ or $x_i \geq \alpha x_i + \beta$. Such a v-bound can be redundant, e.g., if $\alpha = 1$ and $\beta = 0$. The v-bound can also prove infeasibility of the problem, e.g., if $\alpha = 1$ and $\beta < 0$ for the case of an upper bound. As a third possibility, the v-bound may be redundant within an interval of values of x_i, but contradictory for all other values. Consider for example the v-bound $x_i \leq 2x_i + 4$, which is contradictory for $x_i < -4$. By analyzing cycles in the variable bound graph, SCIP can apply the valid bound change $x_i \geq -4$. If this bound is not applied, it may happen that SCIP decreases the upper bound to -5 in branching. This bound could then be decreased to -6 by domain propagation based on the variable bound graph or also the constraints from which the v-bounds were extracted originally. After the decrease to -6, further decrease it to -8 by propagation is possible. This process of decreasing the upper bound can continue until either infeasibility is proven by passing the lower bound of the variable, some limit on the number of propagation rounds is reached, or the upper bound reaches a value of minus infinity. At that point SCIP will also conclude that there is no valid value for this variable and the current sub-problem is infeasible. Thus, detecting the valid bound change at the beginning of the solving process by analyzing the variable bound graph is preferred.

$$\begin{aligned} x - 2y &\leq 3 \quad (1)\\ x + 2z &\leq 2 \quad (2)\\ x + 3y &\leq 6 \quad (3) \end{aligned}$$

(a) Constraint set.

$$\begin{aligned} x &\leq 2y + 3 && (1a)\\ y &\geq \tfrac{1}{2}x - \tfrac{3}{2} && (1b)\\ x &\leq 2 - 2z && (2a)\\ z &\leq 1 - 0.5x && (2b)\\ x &\leq 6 - 3y && (3a)\\ y &\leq 2 - \tfrac{1}{3}x && (3b) \end{aligned}$$

(b) V-bounds.



(c) Variable bound graph.

Figure 1: Example of a variable bound graph.

The topological ordering is computed by a depth-first search in the variable bound graph. During this search, if one of the outgoing arcs of the current node leads to a predecessor of the node in the depth-first search tree, a cycle is detected.

Let $(n_1, \ldots, n_k)$ be such a cycle where n_i is either $lb(x_{j_i})$ or $ub(x_{j_i})$ and $n_1 = n_k$. Each arc (n_i, n_{i+1}) for $i = 1, \ldots, k-1$ represents a v-bound:

1. $x_{j_{i+1}} \geq \alpha_i\, x_{j_i} + \beta_i$ if $n_{i+1} = lb(x_{j_{i+1}})$
2. $x_{j_{i+1}} \leq \alpha_i\, x_{j_i} + \beta_i$ if $n_{i+1} = ub(x_{j_{i+1}})$.

Note that this implies n_i and n_{i+1} represent the same type of bound (lower or upper) if $\alpha_i > 0$, and different bound types (one lower, one upper) if $\alpha_i < 0$. Without loss of generality, let $n_k = lb(x_{j_k})$ in the following.

The v-bound coefficients related to the arcs of the cycle can be aggregated in the following way:

1. $\alpha^1 := \alpha_{k-1}$, $\beta^1 := \beta_{k-1}$
2. $\alpha^i := \alpha_{k-i}\alpha^{i-1}$, $\beta^i := \beta_{k-i} + \alpha_{k-i}\beta^{i-1}$ for $i = 2, \ldots, k-1$.

Then, the following v-bounds are valid:

$$x_{j_k} \geq \alpha^i x_{j_{k-i}} + \beta^i \text{ for all } i = 1, \ldots, k-1, \tag{2}$$

which can be seen as follows. Let $x_{j_k} \geq \alpha_{k-1} x_{j_{k-1}} + \beta_{k-1}$ be the v-bound corresponding to the last arc in the cycle. Now, if $n_{k-1} = lb(x_{j_{k-1}})$, the second-last arc corresponds to $x_{j_{k-1}} \geq \alpha_{k-2} x_{j_{k-2}} + \beta_{k-2}$. This gives a lower bound on $x_{j_{k-1}}$ and since both nodes n_k and n_{k-1} represent lower bounds $\alpha_{k-1} > 0$ holds. Therefore, when replacing $x_{j_{k-1}}$ in the first relation according to the second, this gives the following lower bound relation:

$$\begin{aligned} x_{j_k} \geq \alpha_{k-1}(\alpha_{k-2}\, x_{j_{k-2}} + \beta_{k-2}) + \beta_{k-1} &= \alpha_{k-1}\,\alpha_{k-2}\, x_{j_{k-2}} + \alpha_{k-2}\,\beta_{k-2} + \beta_{k-1}\\ &= \alpha^2\, x_{j_{k-2}} + \beta^2. \end{aligned} \tag{3}$$

On the other hand, if $n_{k-1} = ub(x_{j_{k-1}})$, the second-last edge corresponds to $x_{j_{k-1}} \leq \alpha_{k-2} x_{j_{k-2}} + \beta_{k-2}$. This is an upper bound on $x_{j_{k-1}}$ but now both nodes n_k and n_{k-1} represent different bound types and thus $\alpha_{k-1} < 0$, so (3) is still valid. Applying this argument repeatedly proves validity of the aggregated v-bounds (2).

Since $x_{j_1} = x_{j_k}$, the v-bounds (2) for $i = k-1$ gives the following relation:

$$x_{j_k} \geq \alpha^{k-1} x_{j_1} + \beta^{k-1}. \tag{4}$$

If $\alpha^{k-1} = 1$, this boils down to $0 \geq \beta^{k-1}$ and infeasibility is proven for the case $\beta^{k-1} > 0$. On the other hand, if $\alpha^{k-1} < 1$, $\frac{\beta^{k-1}}{1-\alpha^{k-1}}$ is a valid lower bound of x_{j_k}. Finally, if $\alpha^{k-1} > 1$, $\frac{\beta^{k-1}}{1-\alpha^{k-1}}$ is a valid upper bound of x_{j_k}.

If $n_k = ub(x_{j_k})$, the inequality sign in (4) is changed from $\geq$ to $\leq$ and thus infeasibility is proven for $\alpha^{k-1} = 1$ and $\beta^{k-1} < 0$ and otherwise, $\frac{\beta^{k-1}}{1-\alpha^{k-1}}$ is a valid upper bound of x_{j_k} if $\alpha^{k-1} < 1$, and a valid lower bound if $\alpha^{k-1} > 1$.

Note the argument to derive valid bounds only relies on the fact that $x_{j_1} = x_{j_k}$, the stricter condition $n_1 = n_k$ is not needed. Therefore, the algorithm is also applied for paths with start- and end-node corresponding to the same variable, but representing two different bound types.

The development of this feature was motivated by a toy instance where domain propagation repeatedly tightened the bounds of two variables until the bounds reached infinity. By using the described cycle detection, infeasibility of the respective sub-tree was proven immediately. On standard MIP benchmarks, however, the cycle detection does not find any reductions. Although cycles are identified within the variable bound graphs of some problems, they only provide redundant bounds on the involved variables. Additionally, for performance reasons, SCIP does not build the complete variable bound graph including clique information (see the next section for a definition of cliques). While building the complete variable bound graph would lead to detecting more cycles in the problems, it poses too large a computational overhead. Specifically, the depth-first search performed in the variable bound graph when computing the topological ordering is computationally expensive. For these reasons, cycle detection is currently disabled by default, but it can be enabled by setting the parameter `propagating/vbounds/detectcycles` to `TRUE`.

2.1.2 Connectivity of the conflict graph for clique partitioning

An important global structure of a CIP P containing binary variables is the *conflict graph* G^{conf}. Its use for deriving cutting planes and propagating linear constraints for MIP has been investigated in Atamturk et al. [11]. The conflict graph is defined as the undirected $G^{\text{conf}} := (B \cup \bar{B}, E)$, where B denotes the binary variable indices, $\bar{B}$ the set of complemented binary variables

$$\bar{B} := \{\bar{\jmath} \ : \ j \in B, x_{\bar{\jmath}} = 1 - x_j\},$$

and E denotes the edge set $E := \{\{i,j\} \subseteq B \cup \bar{B} \ : \ x_i + x_j \leq 1 \text{ is valid for } P\}$. The edge set of G^{conf} represents all pairs of binary variables (and their negations), which cannot be set to 1 simultaneously in any feasible solution for P. Note that $\{j, \bar{\jmath}\} \in E$ for all $j \in B$, i.e., there is always an edge between a binary variable x_j, $j \in B$, and its complement $x_{\bar{\jmath}}$.

We call every complete subgraph with node set $C \subseteq B \cup \bar{B}$ a *clique*. A clique of the conflict graph is sometimes referred to as generalized upper bound [11]. It is important to know that SCIP does not store the edges of G^{conf} explicitly, but rather stores a list of cliques of G^{conf}, where each clique is represented as a list of involved vertices. This has advantages in the memory consumption of the conflict graph storage. However, the downside of this representation is the computational cost for querying if a pair of binary variables $\{i,j\} \in B \cup \bar{B}$, $i \neq j$ is connected by an edge in the conflict graph. In the worst case, such a query requires to iterate over the list of cliques of i and j combined.

For a given set of variables $A \subseteq B \cup \bar{B}$, a *clique partition* $\mathcal{A} = \{A_1, \ldots, A_k\}$ is a partition of A such that every A_i is a clique. Clique partitions play an important role for certain presolving

and propagation algorithms. An example is the knapsack constraint handler of SCIP that uses clique partitions for a stronger propagation as described by Atamturk et al. [11].

SCIP provides a greedy clique partitioning algorithm for a set of variables, which considers the connectivity of G^{conf} as of this release. Indeed, two variable indices $i \neq j$ cannot appear in the same part A_k of any clique partition, if they are in different connected components of the conflict graph.

The modified partitioning process is presented in Algorithm 1. The modification consists of only one line of pseudo code, namely line 2. Given connected component labels for every index $j \in A$, a linear time bucket sort algorithm is used to group variables together that belong to the same connected component of G^{conf}. In previous versions, SCIP simply used the entire set $A = A^c_1$, i.e., $s = 1$.

In order to appreciate the modification, it must be noted that the operation in line 12 requires $|A_k|$ checks whether an edge is contained in E in the worst case, which is itself an expensive operation due to the representation of the conflict graph. A threshold t^{comp} ensures that the algorithm does not exhibit quadratic worst behavior if no nontrivial partition of a component exists. Therefore, SCIP uses a hard-coded limit of $t^{\text{comp}} = 10^6$. Thus, the limit can only affect connected components containing at least $|A^c_l| \geq 1000$ variables. When Algorithm 1 reaches the threshold in line 9, the remaining variables are appended as one-element cliques to the returned clique partition $\mathcal{A}$.

This approach has been tested on a special test set of 13 instances from a line planning application [26], which combine very long knapsack inequalities (involving several thousand variables) and set-packing inequalities that do not intersect. Here, the described modification saves 35% of presolving time, and increases the average number of successful calls to knapsack domain propagation by 29%. Before the use of connectedness for clique partitioning, almost all variables would have been partitioned into singleton cliques, thereby weakening the propagation routines.

Algorithm 1: Clique partitioning algorithm of SCIP

Input: Conflict graph $G^{\text{conf}} = (B \cup \bar{B}, E)$, a set of binary variables indexed by $A \subseteq B \cup \bar{B}$, limit $t^{\text{comp}} \geq 1$
Output: A clique partition $\mathcal{A} = \{A_1, \ldots, A_k\}$ of A

```
1  A ← ∅
2  Compute connected component partition A^c_1, ..., A^c_s of A
3  k = 0
4  foreach l = 1, ..., s do
5      Q ← A^c_l
6      t ← 0
7      foreach i ∈ Q do
8          k ← k + 1, A_k ← {i}, t ← t + 1
9          if t < t^comp/|A^c_l| then
10             Q ← Q \ {i}
11             foreach j ∈ Q do
12                 if {j', j} ∈ E ∀j' ∈ A_k then
13                     A_k ← A_k ∪ {j}, Q ← Q \ {j}, t ← t + 1
14         A ← A ∪ {A_k}
15 return A
```

2.1.3 Improved handling of independent components

The components presolver [66] has proven to be very successful in improving the solver performance for problem classes such as supply network planning problems. It exploits a decomposable structure within the problem, namely completely independent sub-problems. For example, in supply network planning problems a decomposable structure arises when different regions or products do not interfere with each other. Given such a structure, the exponential nature of a branch-and-bound search suggests that solving these sub-problems individually should be preferred to solving the problem as a whole. Indeed, computational experiments by Gamrath et al. [66] show significant performance improvements.

The components presolver identifies independent sub-problems and tries to solve them to optimality during presolving using a sub-SCIP, which is an auxiliary SCIP instance to which (a part of) the problem is copied. To this end, it constructs the connectivity graph of the MIP where each node represents one variable and each constraint is represented by a set of edges, connecting the first variable in the constraint with all other variables of the constraint. This representation ensures that all variables in the constraint are within one connected component of the graph while keeping the number of edges in the graph small. The connected components of the graph can be determined efficiently using a depth-first-search and each component then represents one independent sub-problem.

In many cases, near-optimal solutions are suitable in practice for hard problems since the runtime to prove optimality may be too large. Therefore, solving one component to optimality after the other should be avoided in practice if the solving process of single components tends to take a significant amount of time. Thus, only sub-problems that do not exceed a given upper bound on the number of discrete variables are processed by the presolver. If there are sufficiently small sub-problems, they are copied to a sub-SCIP, which is then solved with a node limit to further limit the effort spent in the presolver. If the sub-SCIP was solved to optimality, the respective variables in the original problem are fixed to the optimal values and the related constraints are deleted.

SCIP 4.0 introduces the components constraint handler to extend this concept by providing functionalities to also benefit from larger independent components that cannot be solved to optimality in a short amount of time. First, however, it fully replaces the components *presolver*. A presolving callback is implemented in the constraint handler that is executed during the presolving process and provides the same functionality as the original presolver. Additionally, the components constraint handler allows sub-problems to be solved individually that became independent during the solving process due to branching decisions. For this, (locally) fixed variables are disregarded when checking the connectivity of the problem.

If the problem at one node in the branch-and-bound tree can be divided into independent sub-problems, the node is processed by solving them individually. The optimal solution for this branch-and-bound node is then constructed by merging the solutions of the sub-problems. Other than in presolving, the disconnectivity information should be exploited also for larger sub-problems that cannot be expected to be solved to optimality in a short amount of time. Therefore, the sub-problems are not solved to optimality one after the other. Alternatively, the main solving process may solve other open nodes from the branch-and-bound tree while delaying the solving of subproblems from the components constraint handler.

The components detection is performed in the propagation callback of the `components` constraint handler. If the problem decomposes at the current node, a sub-SCIP is created for each of the components. Additionally, a `components` constraint is attached to the current node to store the sub-SCIPs together with additional information about the decomposition. The general behavior afterwards is the following: rather than processing the node and enforcing the LP

solution by branching if needed, the solving process of one of the sub-SCIPs is continued. In the first call of a sub-SCIP, only its root node is processed. In later calls, the node limit is increased and the gap limit is set to half of the current gap of that problem. After a sub-SCIP was solved or a limit was reached, the dual bound of the current node is updated (if dual bounds for all sub-problems have already been computed) and a new primal solution is constructed from the best solutions found so far in all subproblems. Then, the current node in the main SCIP is postponed. This means that it is put back into the queue of open leaf nodes and will be processed again later. When the node is processed again, the attached `components` constraint signals that a decomposition took place at this node. Therefore, the next sub-SCIP is selected whose solving process is then continued.

When there are multiple open sub-problems at a node, those with a high absolute gap are preferred. However, during the main solving process SCIP tries to keep the number of calls for each sub-SCIP balanced. When all sub-SCIPs related to a decomposition at a node were solved to optimality, this node is pruned. The same happens if the dual bound obtained through the sum of the sub-SCIPs dual bounds exceeds the cutoff bound in the main SCIP.

Note that creating a number of sub-SCIPs for the individual components causes some overhead that might outweigh the benefits obtained by solving several smaller problems rather than one large one. Therefore, some parameters are provide to adjust the handling of components. First, the maximum depth up to which the detection is applied can be specified. Nodes far down in the tree can often be expected to be solved without too much additional effort, so the overhead of creating the sub-SCIPs should be avoided. It also happens regularly that after some branching decisions, a few variables become disconnected but do not complicate the current problem, for example because they are integral in each optimal LP solution. The overhead of creating a sub-SCIP in such a case is avoided by enforcing a minimum size for a sub-SCIP to be created. Multiple smaller components will be merged to a larger one if this achieves the threshold. Note that more tuning and research is needed in this context.

Computational results have been performed using a set of supply chain instances and general MIP instances from MIPLIB. On the supply chain models, components handling during the branch-and-bound search can drastically improve the solving performance. This holds in particular for large supply network planning problems where some of the components are too large to solve in presolving. The performance improvement from the components constraint handler has been observed on one particular instance that can be split into 66 independent sub-problems at the root node. The use of this constraint handler helps to reduce the optimality gap to less than 1 % within a few seconds. When the components detection is restricted to small sub-problems in presolving, the gap after 20 minutes is still larger than 60 %.

General MIP instances on the other hand feature decomposable structures less often. If a decomposable structure is identified, the independent components are small enough most of the time to solve them to optimality during presolving. Therefore, component handling during the tree search introduces computational overhead for these instances with no performance improvement. As such, the new component handling is disabled by default and only small components are solved during presolving.

2.1.4 Improvements of propagation loop and event handler system

An issue sometimes identified in the development of pre-root heuristics is the unexpected change in solver behavior after their call. In particular, even if a heuristic did not find a new solution or update additional data, such as conflicts or inference scores, its call altered the branch-and-bound tree search.

The reason why this occurs was identified as follows: The heuristics were executed at the

beginning of the node processing, before domain propagation at the node was called. Additionally, they often start the probing mode of SCIP to create a temporary dive in the tree where at each auxiliary node, some variables are fixed and those changes are propagated afterwards. This led to the issue that the domain propagation calls within the heuristic changed internal flags of the constraint handlers, stating that domain propagation or specific propagation algorithms were already performed for a constraint. The plugin concept of SCIP does not allow for an automatic reset of these flags when the probing mode is finished and domain propagation at the root node is performed. As a consequence, the internal flags did not always correctly reflect the need to propagate a constraint. Thus, root node propagation might incorrectly not propagate some constraints.

This issue was fixed by two changes to the main solving loop in SCIP. First, the constraint handlers now use the SCIP methods `SCIPmarkConsPropagate` and `SCIPunmarkConsPropagate` more frequently. These methods provide a unified scheme for marking/unmarking a constraint to be processed in the next propagation round, for example if a bound of one of the variables in the constraint is tightened. By using these methods rather than internal flags in the constraint data, SCIP 4.0 can ensure that the propagation status of a constraint is not lost during intermediate propagation calls in probing mode. To this end, all marked constraints are buffered when probing is started and this information is restored when probing ends. Second, even if no information is lost, SCIP first propagates the general root propagation plus some additional changes at an auxiliary probing node. After the probing heuristic finished, everything is reset and almost the same propagation is done again to apply the changes at the root node. This inefficient behavior is overcome by performing the first domain propagation call at each node now before calling the first heuristic at this node.

Additionally, the evaluation of whether performing another presolving round is promising was improved so that another propagation round is now triggered more often. In particular, successful propagation calls with timing `DURINGLPLOOP`, which are called during the separation loop, will trigger another propagation round with timing `BEFORELP`, which mainly includes non-LP-based algorithms like activity-based bound tightening. The interaction of `BEFORELP` and `AFTERLPLOOP` propagators was also improved. The latter propagators typically execute more expensive algorithms that need information from an optimal LP relaxation. If the LP solution changed after the last `AFTERLPLOOP` propagation call, another propagation round with this timing will be performed directly before the `BEFORELP` propagators, which can further tighten domains.

Finally, propagation was improved by a change to the event handling system. Most constraint handlers make use of event handlers in order to be informed when a bound in one of their constraints was changed. This information is used to update activities or mark the constraint for propagation. When jumping to a node in a different sub-tree, SCIP first traces the path in the tree to the common ancestor node, thereby relaxing all bounds that were previously tightened at the previous focus node. After that, nodes are activated on the path to the new focus node. To reduce the event handling effort, the corresponding bound change events are not processed one after the other, but rather buffered and only the final bound change event for each variable is processed. This results in at most one event being handled per bound of a variable. In many cases, no event has to be processed for a variable at all, for example if it was fixed to the same value in both sub-trees. This reduces the effort needed for activity updates and avoids unnecessary propagation calls. If a node was previously propagated, all bound tightenings are re-applied when it is once again activated. So there is no need to repropagate this node. However, this is not the case for the new focus node, where all bound tightenings compared to its parent node need to be taken into account when deciding which constraints should be propagated. With SCIP 4.0, buffering of bound changes is only done for the switch from the old focus node to the parent of the new focus node. All bound changes applied at the new focus node trigger an event.

Computational results. Together these changes improve SCIP by applying successful domain propagation more often and more consistently. Computational experiments to evaluate the changes were performed on the MMM test set which is the union of the last three MIPLIB versions: MIPLIB 3 [24], MIPLIB 2003 [5], and the benchmark set of MIPLIB 2010 [85]. We ran SCIP with five different random seeds on each instance, once with and once without the changes described in this section. The random seeds are used to reduce the impact of performance variability [85] and get more significant results. Each individual instance is counted as solved only if it was solved within the time limit of two hours with all five seeds. The solving time for an instance is the average over the solving time with the five seeds. The updates described in this section allow to solve two more instances and reduce the solving time by 2 %. The shifted geometric mean over the number of processed branch-and-bound nodes was even reduced by 5 % for the set of instances solved by both variants. However, more importantly, these changes help reduce performance variability and the unintended side effects of including newly developed plug-ins, which cannot be measured directly. An indication for the reduced variability is the fact that the relative difference between the shifted geometric means of the solving time for the fastest and the slowest seed is reduced from 7.6 % to 5.5 %

2.1.5 Enforcement of relaxation solutions

An important part of a branch-and-bound approach is solving continuous relaxations of the CIP or MINLP. There are two main uses of continuous relaxations in branch-and-bound algorithms. First, the bound computed by the continuous relaxation can be used to prune the node, given a better known solution. Second, the solution can be enforced, meaning that it is used for branching or to compute cutting planes. While the usage of the computed bound was implemented for both the LP and general relaxations in previous SCIP versions, only the LP solution was enforced. SCIP 4.0 has been extended to include the possibility to enforce the solution of arbitrary relaxators and use their tighter relaxations to compute stronger cutting-planes or make better branching decisions. This also allows user-written relaxators to be combined with any type of constraint handlers implemented in SCIP—even without solving any LPs—since the constraint handlers can now work on the relaxation solution. In this section, the usage of non-LP relaxations in a branch-and-bound context and its potential for enhancing the solving process is discussed and the implementation of the relaxation enforcement in SCIP is explained.

While the enforcement of relaxation solutions may be useful for general CIPs, it is most interesting for convex and nonconvex MINLPs. The usual approach to solve nonconvex MINLPs is to combine a spatial branch-and-bound or branch-and-reduce algorithm, as introduced by Ryoo and Sahinidis [120], with convex under- and concave over-estimators. The most important decision in the design of such algorithms is the choice of over- and under-estimators. They can either be chosen as linear functions—like in SCIP with default plugins—or as general nonlinear convex or concave functions like in the α-BB approach [8, 9]. The advantages of solving linear relaxations are the efficiency of LP solvers and the availability of sensitivity information which can be used for inferring domain reductions, see for example Burer and Letchford [32].

The disadvantage of the LP-based approach is that an arbitrary number of spatial branchings or cutting planes may be needed to enforce even a single convex constraint using linear functions only. Therefore, it may still be worthwhile for many applications to solve nonlinear relaxations in some or all nodes of the branch-and-bound tree of a convex or nonconvex MINLP to compute stronger dual bounds, make better branching decisions and create stronger cutting planes already in the first enforcement rounds, as shown for example by the numerical experiments of Bonami et al. [25]. Furthermore, for specific applications special types of relaxations like semidefinite relaxation as in Buchheim and Wiegele [31] and Burer and Vandenbussche [33] or even mixed-

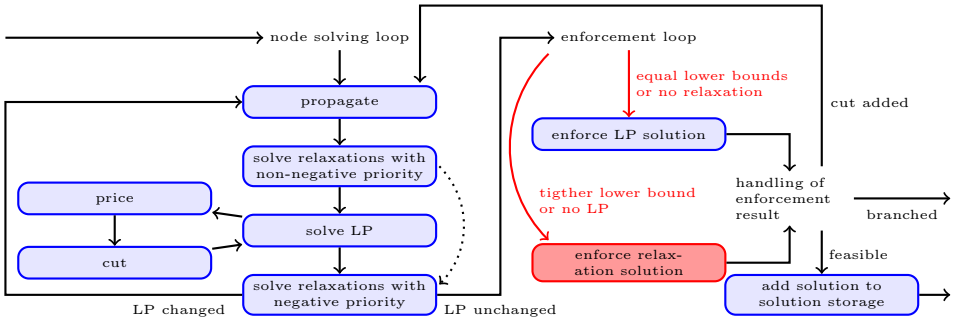


Figure 2: Node processing loop in SCIP 4.0 with differences to previous version

integer linear relaxations like in Geißler et al. [68] may be available.

Feature description. The core of SCIP 3.2.1 only uses the lower bounds computed by relaxator plugins, while the relaxation solution is only used if the relaxator itself, manually, tries to add it to the solution storage or adds external branching candidates. In SCIP 4.0 the best relaxation solution is stored for comparison with the LP solution. The relaxation solution is enforced if it's objective value is strictly greater (for a minimization problem) than that of the LP or if the LP was not solved at the current node. Note that if the optimal objective of the LP is equal to the objective of the best relaxation, the LP solution will be enforced.

It is necessary to restrict this process to those relaxators that include all linear constraints in their relaxation. This ensures that added cuts are obeyed by the relaxation and the previous optimal solution is cut off when resolving the relaxation after adding corresponding cuts. For this purpose, the additional parameter `RELAX_INCLUDESLP` has been added to the properties of relaxation handlers. If this flag is set to `false`, or if the relaxation handler only provides a lower bound but no solution, the behavior is the same as with SCIP 3.2.1.

In case the relaxation solution is chosen for enforcement, the enforcement-loop will try to cut off the relaxation solution if it violates some constraint. If all constraints are fulfilled but some integral variables take fractional values, these will be added as external branching candidates. Finally, if the solution is feasible for all constraints including integrality, the relaxation solution is added to the solution storage. For an overview of the updated node processing loop with and without solving LPs see Figure 2, with the main difference to SCIP 3.2.1 being the additional option of enforcing the relaxation solution.

2.1.6 Partial Solutions

In practice it is often possible to guess problem specific solution values for a subset of the variables. One such example is setting all integer variables to zero. Guessing solution values for only a subset of variables can be interpreted as a partial solution. A solution $\hat{x}$ is called partial, if $\hat{x}_i \in [l_i, u_i]$ for all guessed solution values, and $\hat{x}_i = \perp$ otherwise, which means that the variable has an unknown solution value. These guesses can be used to produce primal solutions heuristically.

In SCIP 3.2.1 and previous versions, however, the task of extending such partial assignments to a complete and feasible solution that could be added to SCIP was left to the user. SCIP 4.0 now provides the possibility to read in (via the console) and create (via the API) solutions where only a subset of variables is fixed to a solution value. At the beginning of a solve, before

presolving the given problem, a new primal heuristic (called `completesol`, see 2.3.2) will be executed if partial solutions are present.

In contrast to the usual SCIP solution format `.sol`, where each variable that is not listed is assumed to be fixed to zero, the new solution format `.mst` (MIP start) introduced in SCIP 4.0 describes a partial solution where all non-listed variables are assumed to be unfixed, i.e., the solution values are unknown. However, both formats can describe the same solutions as is the case in the following example.

`.sol` format:	`.mst` format:
x_1 1	x_1 1
x_4 0.5	x_2 0
x_5 unknown	x_3 0
	x_4 0.5

In order to allow for partial solutions defined in the `.sol` format, the format has been extended. Specifically, a variable without a "guessed" solution is labeled as unknown. As such, in SCIP 4.0 each variable stated in a `.sol` file can take either a real value, $\pm$`inf`, or `unknown`.

In addition to the new functionality of reading partial solutions, SCIP 4.0 provides a new interface method `SCIPcreatePartialSol` that creates a partial solution. Analogous to complete solutions, partial solutions can be added by calling `SCIPaddSolFree` or `SCIPaddSol` before the solving process commences. As for complete solutions, both methods add the solution to the internal storage, `SCIPaddSolFree` additionally frees the solution pointer afterwards. Moreover, SCIP 4.0 provides the possibility to write only the integral part of the best known solution to a file. Such a solution is also known as a *MIP start*, which can be written and read by other MIP solvers like Gurobi, Xpress, and CPLEX.

2.1.7 Randomization in SCIP

Random numbers have a strong influence on the solution process of a modern MIP solver. This is due to the multitude of locations where (pseudo) random numbers are used within the solver. For example, random numbers can be used if a tie needs to be broken when ordering variables or constraints. Moreover, random numbers are also used to reduce numerical troubles by perturbing objective coefficients. Since changing the random seed of a MIP solver leads to a different solving behavior it can also be used to simulate testing on a much larger set of instances. Averaging the results for each instance over all used random seeds leads to more robust results when measuring the overall performance by reducing the impact of outliers. Alternatively, a test set can be enlarged by permuting variables and constraints of a given problem. However, testing and tuning without changing the seeds used within a random number generator can lead to the over tuning of parameters in a MIP solver. One negative outcome of over tuning can be large performance variability after the addition or removal of plugins.

In previous versions of SCIP it was not possible to easily change the random behavior of the solver. To remedy this, SCIP 4.0 provides a new parameter class `randomization` to modify the random numbers used during the solving process. This new parameter class contains three different parameters, which can be found in Table 1. Changing the new parameter `randomseedshift` will affect all random numbers used in the plugins by shifting the predefined initial random seed. Beside changing the random behavior of the SCIP plugins itself, the random seed of the linked LP solver can also be changed (`lpseed`). This parameter is also affected by changing the random seed shift. The random seed that is used to permute the variable and constraint ordering of a

Table 1: Overview of all global parameters related to randomization in SCIP 4.0.

parameter	range	default	description
`lpseed`	$[0, 2^{31}]$	0	Random seed for LP solver, e.g. for perturbations in the simplex.
`permutationseed`	$[0, 2^{31}]$	0	Seed value for permuting the problem after reading/transformation.
`randomseedshift`	$[0, 2^{31}]$	0	Global shift of all random seeds in the plugins and the LP random seed.

given problem instance (`permutationseed`) will not be affected by changing the random seed shift.

In addition to providing convenience functions that allow changing the random behavior, SCIP 4.0 comes with a new random number generator. Instead of using the C library function `rand()`, the KISS random number generator [98] is used. In comparison to `rand()`, KISS has a much larger period ($> 2^{124}$) and is able to pass statistical tests, such as the BigCrush-Test from the TestU01 library [89]. This has no impact on the performance of SCIP, but increases pseudo random number generation stability.

Given the heavy reliance on random number in SCIP, the arithmetic operations used by a random number generator have an impact on the overall solving performance. Compared to other random number generators that pass statistical tests of randomness, like Mersenne-Twister, KISS uses only simple arithmetic operations like multiplication, addition, and bitwise `XOR`.

2.1.8 Hash tables in SCIP

Hash tables are a widely used data structure where elements are stored in an array by computing a hash value from their keys. This allows elements from a large key universe to be stored in a much smaller array, while accessing the elements by their key still requires amortized constant time. Hash tables are used throughout the SCIP code, for example when finding duplicate constraints or cliques. The performance of the hash table implementation can have a high impact on the presolving time as there are instances where several millions of cliques are found. A hash table allows to determine the duplicates using pairwise comparisons only on the elements that have an equal hash value. When different elements have the same hash value, this is called a collision. A high quality hash function should be fast to compute and distribute the keys well, ideally as if the hash values where chosen uniformly at random.

One improvement for the hash tables in SCIP 4.0 regards the hash function. Previously the size of the array was chosen to be a prime number and the hash function was solely a modulo operation with this size, whereas SCIP 4.0 uses a multiply-shift hash function [46] given by

$$h(x) = \frac{(ax \bmod 2^{64})}{2^{64-l}},$$

where $a \in [1, 2^{64} - 1]$ is some odd integer and $l \in \mathbb{N}$ is the desired bit length of the output. This hash function is fast to compute and distributes the keys well [132, 117]. In fact it was shown to be universal, i.e., for a chosen uniformly at random and arbitrary $x \neq y$ the probability for $h(x) = h(y)$ is at most $\frac{2}{2^l}$ [46].

When hashing from a larger universe of keys into the indices of a smaller array collisions cannot be avoided and need to be resolved by the hash table implementation. The hash table implementation of prior SCIP versions used *chaining*, a common strategy for resolving a collision.

When using this strategy each slot of the array stores a linked list containing all elements that where hashed to this index. While this is easy to implement, a series of pointers must be followed when looking for an item in one of the linked lists. Due to cache effects in modern microprocessors, such scattered memory accesses are far more expensive than accesses to contiguous memory locations. Another strategy for collision resolution is *open addressing*. Here a collision is resolved by storing an element at a different position than the one it was hashed to, according to some rule. A simple rule that highly compatible with the CPU caches is called *linear probing*. As the name suggests, the array is scanned linearly until an empty slot is found.

In SCIP 4.0 the old implementation using chaining was replaced by an implementation of Robin Hood hashing [36] with linear probing. The name stems from the insertion algorithm: if an element is encountered that has a better position than the element being inserted—closer to the position it was hashed to—the position will be "stolen" from this element and given to the new one. In that case the insertion continues with the element until it either takes the position of another element or finds an empty slot.

The Robin Hood insertion strategy reduces the variance and the worst-case of the required search length for a lookup in the hash table and gives good performance even for high occupancy. This allows SCIP 4.0 to use considerably smaller hash table sizes than before. A comprehensive experimental comparison of hashing methods by Richter et al. [117] supports the algorithmic choices for the new hash table implementation. They found the multiply-shift hash function to be the best trade-off between distribution quality and throughput in practice. Among the collision resolution schemes they compared, Robin Hood hashing with linear probing was always among the top performers and in many cases the best method.

2.2 Presolving

Presolving applies a collection of algorithms in an attempt to reduce the size of the given problem or improve its formulation. Achterberg et al. [4] have recently presented an overview that covers a broad range of presolving techniques for solving mixed-integer programs. In this section, two new presolving techniques included in SCIP 4.0 are described. Further, the coordination of the presolving methods has been improved in SCIP 4.0 with addition of the new presolving level `FINAL`.

Extended stuffing presolving. Singleton variable stuffing [66] was previously implemented using the matrix module of SCIP [63]. This introduced some overhead that is now avoided by moving this presolving step into the linear constraint handler.

Additionally, a related presolving step was added to the linear constraint handler, called *single variable stuffing*. This presolving step is not restricted to continuous variables and can also be applied to non-singletons, as long as the variable locks allow this.

Before going into detail about the new presolving step, let us shortly review the concept of variable locks as introduced by Achterberg [1].

Definition 2.2.1 (Variable locks)**.** *Let x_j be a variable in problem (1). A constraint $g_i(x) \leq 0, i \in \mathcal{M}$* down-locks *(*up-locks*) x_j if there exists a solution $\bar{x}$ with $g_i(\bar{x}) \leq 0$ and $\varepsilon > 0$ such that $g_i(\bar{x}') > 0$ for $\bar{x}' = \bar{x} - \varepsilon e_j$ ($\bar{x}' = \bar{x} + \varepsilon e_j$). The number of constraints that down-lock (up-lock) a variable is called the number of down-locks (up-locks) of that variable.*

Single variable stuffing is a presolving step that investigates a single constraint. Let a constraint $\sum_{j \in J} a_j x_j \geq b$ be given with non-negative variables $x_j, j \in J$, coefficients $a_j > 0$ for $j \in J$, right hand side $b > 0$, and objective coefficients $c_j \geq 0$ for all variables $x_j, j \in J$. Since all variables have a non-negative coefficient, setting all variables to 0 would be optimal with respect

to the objective function but infeasible for the problem due to the investigated constraint. However, if one variable x_k with smallest ratio $\frac{c_k}{a_k}$ suffices to satisfy the constraint, its lower bound can be increased to $\frac{b}{a_k}$ and all other variables $x_j, j \in J \setminus \{k\}$ can be fixed to 0 under the following conditions:

1. $\frac{b}{a_k} \leq u_k$, where u_k is the upper bound of the variable,
2. x_k has no up-locks,
3. all $x_j, j \in J \setminus \{k\}$ have exactly one down-lock (originating from the constraint that is investigated), and
4. if x_k is an integer variable, then $\frac{b}{a_k} \in \mathbb{Z}$ or $c_k \leq \frac{c_j}{a_j}\left(b - \left\lfloor \frac{b}{a_k} \right\rfloor a_k\right)$ holds for all $j \in J \setminus \{k\}$.

Through the lower bound tightening of x_k, the constraint $\sum_{j \in J} a_j x_j \geq b$ becomes redundant. This presolving step is implemented in SCIP 4.0 and generalized to variables with lower bounds different from 0. Further, it is possible to investigate variables with negative coefficients if they also have non-positive objective coefficients.

Exploiting complementary slackness. The theory of duality for linear programming is well-developed and has been successfully applied in advancing algorithms for solving linear programs. In principle, many of the results for linear programming can be extended to mixed-integer programming, but this has proven difficult. The method described in this section is a particular case where the conveyance of properties from linear programming duality to mixed-integer programming works well.

Consider for $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and variables $x \in \mathbb{R}^n$ the mixed-integer program

$$\min \{c^\top x \,:\, Ax \geq b,\, x \geq 0,\, x_j \in \mathbb{Z}\ \forall j \in \mathcal{I}\}. \tag{5}$$

By applying bound strengthening on the dual of the linear programming relaxation of (5) appropriately, we are able to determine bounds on the dual variables and on reduced costs. The reduced costs may be used to fix variables and the bounds on the dual variables to detect implied equalities in the mixed-integer program (5).

Let us consider one important statement from linear programming duality. If x^* is a feasible solution for

$$\min \{c^\top x \,:\, Ax \geq b,\, x \geq 0\} \tag{6}$$

and y^* is a feasible solution for

$$\max \{b^\top y \,:\, A^\top y \leq c,\, y \geq 0\}, \tag{7}$$

then a necessary and sufficient condition for x^* and y^* being optimal for (6) and (7), respectively, is complementary slackness [125].

To gain valuable insights for problem (5), we are primarily interested in the following implications of complementary slackness

$$\begin{array}{lcl} y_i^* > 0 & \Rightarrow & A_{i\cdot} x^* - b_i = 0, \\ c_j - (A_{\cdot j})^\top y^* > 0 & \Rightarrow & x_j^* = 0, \end{array}$$

where $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}$. An important part of this presolving step is determining the values of y^*. However, solving a linear program introduces a computational burden. So it may be more efficient to alternatively obtain bounds on the dual variables $\bar{\ell} \leq y \leq \bar{u}$. This

is achieved by performing bound strengthening only on the columns of (5) corresponding to continuous variables.

We summarize this idea in the following statement, which was first published in Achterberg et al. [4]. By considering only the columns of the continuous variables x_j with $j \in S := \mathcal{N} \setminus \mathcal{I}$ and applying bound strengthening on $(A_{\cdot S})^\top y \leq c_S$, $y \geq 0$, to get valid bounds $\bar{\ell} \leq y \leq \bar{u}$, the following implications hold:

i) If $\bar{\ell}_i > 0$ for $i \in \{1, \ldots, m\}$, we can convert constraint i into an equation $A_{i\cdot} x = b_i$.

ii) If $c_j > \max\{(A_{\cdot j})^\top y : \bar{\ell} \leq y \leq \bar{u}\}$ for $j \in \{1, \ldots, n\}$, we can fix x_j to its lower bound 0.

This presolving technique typically detects only limited reductions on MIPLIB instances. However, on real-world supply chain instances several thousand of constraints can be turned into equations, which for its part triggers aggregation presolving techniques. Although this presolving technique can fix integer variables, we observe that the majority of fixed variables is of continuous type for supply chain instances.

New presolving level `FINAL`. The concept of presolving levels introduced in SCIP 3.2 (see Gamrath et al. [63]) is extended in SCIP 4.0 by the introduction of the `FINAL` level. Presolvers declaring the level `FINAL` are only called once after all presolvers for the levels `FAST`, `MEDIUM` and `EXHAUSTIVE` did not find enough reductions to trigger another presolving round. The level `FINAL` allows for the implementation of presolve reductions that do not enable further reductions. An example for this is the components presolving, see Section 2.1.3 and [66]. When successfully applied, it fixes all variables of an independent subproblem and deletes the corresponding constraints. Therefore, no reductions in other (independent) parts of the problem can be derived from that result. Consequently, the `FINAL` presolving level is always the last one and never triggers another round of presolving.

Please note that it can not be assumed that a presolver called in this level is the last to perform any reductions, so further cleanup of internal structures in the `PRESOLEXIT` callbacks may still be needed.

2.3 Primal heuristics

Primal heuristics denote auxiliary algorithms aimed at finding good incumbent solutions to support the branch-and-bound procedure. For the release of version 4.0, some new primal heuristics have been added to the battery of about 50 primal heuristic plugins of SCIP. *Graph induced neighborhood search (GINS)* (Section 2.3.1) is a large neighborhood search (LNS) heuristic that uses the distance between variables in the *connectivity graph* for its neighborhood definition. *Completesol* (Section 2.3.2) is a new LNS heuristic designed to complete partial solutions, see also 2.1.6. Furthermore, SCIP comprises a new primal heuristic *LP face* (Section 2.3.3) that searches the optimal face of the LP relaxation, as well as a new constructive pre-root heuristic, *Locks* (Section 2.3.4). Note that these primal heuristics were designed with the primary goal to improve SCIP on MIP problems. For new MINLP-specific primal heuristics in SCIP, see Section 2.6.2.

Numerous primal heuristics for mixed-integer linear optimization have been proposed in the literature, see [19] for an overview and a SCIP-specific list of available primal heuristics. A widely-applied concept is *large neighborhood search (LNS)*. An LNS heuristic for MIP performs the three main steps:

1. Create an auxiliary mixed-integer optimization problem P^{aux}.
2. Solve P^{aux}.

3. Transform solutions found for P^{aux} back to the main problem.

In designing LNS heuristics for MIP there is a lot of freedom regarding the definition of a search neighborhood P^{aux}. The only restriction placed on P^{aux} is that it should contain integer variables. Most proposed LNS heuristics derive their P^{aux} by fixing the domains of a subset $\mathcal{I}' \subsetneq \mathcal{I}$ of the integer variables to values of an incumbent solution at hand. This procedure is formalized in Algorithm 2. The subroutine `LNS-fix` expects a MIP P and a (not necessarily feasible) reference point x^*. The search space is reduced by restricting all values of the variables indexed by $\mathcal{I}'$ to their reference point values. If the reference point is a solution of P, the search is further restricted to solutions that improve the objective function value of x^* by at least ϵ in line 4. Note that the objective cutoff may be rounded down if the objective is known to be always integer. The remaining problem is solved in line 5. Since the solving time of the resulting P^{aux} can take arbitrarily long even for very large sets $\mathcal{I}'$, working limits on the solution process must be applied in order to achieve acceptable running times.

Alternative approaches to define auxiliary LNS problems include the addition of constraints to the original formulation [54], a combination of variable fixings and additional constraints [69], or the use of an alternative objective function [56].

Algorithm 2: Subroutine `LNS-fix`$(P, x^*, \mathcal{I}')$

Input: MIP P, reference point x^*, variable subset $\emptyset \neq \mathcal{I}' \subsetneq \mathcal{N}$ such that $\mathcal{I} \setminus \mathcal{I}' \neq \emptyset$
Output: An improved solution x^{lnsfix}, or **null** if not successful

1 $P^{\text{aux}} \leftarrow P$
2 Add restrictions $\{x_j = x_j^*\}$ to P^{aux} $\forall j \in \mathcal{I}'$
3 **if** *x^* feasible* **then**
4 Add objective cutoff $c^\top x \leq c^\top x^* - \epsilon$ to P^{aux}
5 Solve P^{aux}
6 **if** *P^{aux} has solution x^{aux}* **then**
7 $x^{\text{lnsfix}} \leftarrow x^{\text{aux}}$
8 **else**
9 $x^{\text{lnsfix}} \leftarrow$ **null**
10 **return** x^{lnsfix}

2.3.1 Graph induced neighborhood search

For a given MIP P

$$\min \{c^\top x : Ax \geq b,\ l \leq x \leq u,\ x_j \in \mathbb{Z}\ \forall j \in \mathcal{I}\},$$

let the undirected *connectivity graph* of P be $G_P = (V \cup W, E)$ with $V \coloneqq \{v_1, \ldots, v_n\}$, $W \coloneqq \{w_1, \ldots, w_m\}$ and $E \coloneqq \{\{v_j, w_i\} : A_{ij} \neq 0\}$. The connectivity graph connects the node corresponding to a variable x_j to the nodes corresponding to the rows of A where x_j has a nonzero coefficient. Note that the definition of G_P allows a straightforward generalization beyond linearly constrained problems to arbitrary CPs. In the following, G_P is assumed connected. For two (variable) nodes $v, v' \in V$, let $\mathcal{R}_{v,v'}$ denote the set of all paths connecting v and v'. For a path $R \in \mathcal{R}_{v,v'}$, the number of edges of R is called the *length* $|R|$ of R. The length of a shortest path between v and v' is called the *distance*

$$d(v, v') \coloneqq \min_{R \in \mathcal{R}_{v,v'}} |R|$$

between v and v'. For $v = v'$, the distance is $d(v, v) = 0$. All other nodes $v' \in V$ have an even distance from v because G_P is bipartite—variable nodes are only adjacent to constraint nodes. For $v \in V$, the *k-neighborhood* $\mathcal{N}_k(v)$ comprises all nodes $v' \in V$ with distance at most $2 \cdot k$ from v, and v is called *central node* of $\mathcal{N}_k(v)$.

This heuristics ranks two k-neighborhoods around central nodes $v \neq v'$ by the *root LP potential*: For $k \geq 0$, a root LP solution x^0 and a current incumbent solution x^*,

$$\pi_k(v) \coloneqq \sum_{v_i \in \mathcal{N}_k(v)} c_i \left(x_i^* - x_i^0\right)$$

is called the *root LP potential* of v.

The outline of the proposed graph induced neighborhood search (GINS) primal heuristic is given in Algorithm 3. For a subset $S \subseteq \mathcal{N}$, Algorithm 3 uses the notation $V_{[S]} \coloneqq \{v_i \ : \ i \in S\}$ for the subset of V indexed by S.

Algorithm 3: Rolling horizon algorithm of the GINS heuristic

Input: MIP P with integer variable indices $\mathcal{I}$, incumbent solution x^*, distance k, minimum fraction α of integer variables that must be fixed in the auxiliary problems

Output: An improved solution, or x^* if not successful

```
/* 1. Selection of an initial neighborhood                                  */
Construct the connectivity graph G_P = (V ∪ W, E)
Q ← V_[I]
v^(0) ← null, π^(0) ← −∞
while Q ≠ ∅ do
    Select v ∈ Q uniformly at random
    if 2 ≤ |N_k(v) ∩ V_[I]| ≤ (1 − α)|I| and π_k(v) > π^(0) then
        v^(0) ← v, π^(0) ← π_k(v)
    Q ← Q \ N_k(v)
if v^(0) = null then return x*
/* 2. Solve series of auxiliary problems with increasing distance from v^(0) */
l ← 0, x^(l) ← x*, Q ← V_[I]
while Q ≠ ∅ do
    v^next ← argmin_{v∈Q} d(v, v^(0)), Q ← Q \ {v^next}
    if 2 ≤ |N_k(v^next) ∩ V_[I]| ≤ (1 − α)|I| then
        x^aux ← LNS-fix(P, x^(l), I \ {i : v_i ∈ N_k(v^next)})
        if x^aux ≠ null then
            l ← l + 1, x^(l) ← x^aux, Q ← Q \ N_k(v^next)
        else
            break
return x^(l)
```

Algorithm 3 involves two major stages. The first is the selection of an initial neighborhood. The second is the processing of a series of auxiliary problems as long as improving solutions are found in a rolling horizon fashion on structured problems. Throughout Algorithm 3, the parameter α is used to control the difficulty of the resulting auxiliary problems by restricting the fraction of unfixed integer variables. Neighborhoods with more than $(1 - \alpha)|\mathcal{I}|$ integer variable

nodes are discarded by the conditions in line 6 and line 13. A random sampling procedure is used for the selection of an initial neighborhood. During one sampling step starting from line 4, an integer variable node v is selected uniformly at random, and its k-neighborhood is checked for an appropriate number of nodes corresponding to integer variables. If the root LP potential $\pi_k(v)$ exceeds the root LP potential of the currently selected node, v is selected as initial variable. Regardless of its size or root LP potential, $\mathcal{N}_k(v)$ is dropped from further sampling iterations in line 8.

It is not guaranteed that a suitable initial neighborhood can be found for the selected value of the parameter α. However, if Algorithm 3 reaches line 12 for the first time, the selected node v^{next} is always $v^{(0)}$ whose neighborhood is ensured to match the size requirement because of the selection procedure. Therefore, at least one auxiliary problem is created and solved by calling the subroutine `LNS-fix`. If successful, the procedure then solves a series of auxiliary problems for neighborhoods around central nodes with an ever increasing distance from $v^{(0)}$. By removing the current neighborhood from the set of remaining variables, it is ensured that no central variable from a previous iteration is part of a future neighborhood.

2.3.2 Completion of partial solutions

Since SCIP 4.0 the user has the possibility to add partial solutions (see Section 2.1.6) via the API and the interactive shell. At the beginning of a solve SCIP 4.0 tries to complete all given partial solutions. This is achieved by calling the new LNS heuristic *completesol* before presolving.

Recall that for a given MIP P

$$\min\{c^\top x \,|\, Ax \geq b, l \leq x \leq u, x_i \in \mathbb{Z}\, \forall i \in \mathcal{I}\},$$

a partial solution has the form $\hat{x} \in \mathbb{R}^n$, with $\hat{x}_i \in \{[l_i, u_i], \perp\}$ for all $i = 1, \ldots, n$. The heuristic used to complete a given partial solution is summarized in Algorithm 4. The auxiliary problem used in this LNS heuristic depends on the partial solution $\hat{x}$. First, the domain of all integer variables for which a solution values is given gets reduced, either by fixing or reducing the domain. In a second step the heuristic tries to reduce the domain of all continuous variables with a given solution value. In addition to the domain reduction steps the objective function can be modified to prefer solutions close to $\hat{x}$.

So that not too much time is spent in Line 18 of Algorithm 4 working limits are used. In addition to the standard limits on the number of exploited branch-and-bound nodes or memory consumption, the heuristic will stop if a certain number of improving solutions was found. By default, SCIP 4.0 will stop after 5 (improving) solutions were found. Moreover, SCIP 4.0 will not try to complete a given partial solution if the number of unknown solution values is greater than a parameterized threshold. The optional loop can be activated to enforce solutions that are close to $\hat{x}$, which is disabled by default.

2.3.3 LP face

It is possible that the hyperplane $\{x : c^\top x = c^\top x^{\text{LP}}\}$ contains a solution for a MIP P, but the LP relaxation solution x^{LP} does not satisfy the integrality requirements. In such a case, the dual bound obtained from the LP relaxation solution is already best possible and cannot be further improved by branching. There exist both primal heuristics such as feasibility pump [53] as well as LP methods, for example Section 4.2, which repeatedly solve an LP to attain feasibility. However, if there is no basic solution that is feasible for P, it might not be enough to rely on such LP-based methods.

The *LP face* heuristic is an LNS approach that restricts the neighborhood to the optimal LP face of P. Therefore, all nonbasic (integer and continuous) variables with positive reduced

Algorithm 4: Completesol heuristic

Input: MIP P, partial solution $\hat{x}$, index set of all variables $\mathcal{N}$, index set of integer variables $\mathcal{I}$, scalar for bound-widening $\beta \in [0, 1]$.

Output: An feasible solution x^{compl}, or **null** if not successful.

$P^{\text{aux}} \leftarrow P$
for $i \in \mathcal{I}$ *with* $\hat{x}_i \neq \perp$ **do**
  if $\hat{x}_i \in \mathbb{Z}$ **then**
    Add restriction $\{x_i = \hat{x}_i\}$ to P^{aux}
  else
    /* Allow ranges with size up to four to increase degree of freedom */
    Add restrictions $\{\max\{l_i, \lfloor \hat{x}_i \rfloor - 1\} \leq x_i \leq \min\{u_i, \lceil \hat{x}_i \rceil + 1\}\}$ to P^{aux}
for $i \notin \mathcal{I}$ *with* $\hat{x}_i \neq \perp$ **do**
  Add and propagate domain reduction $\{\max\{l_i, \lfloor \hat{x}_i \rfloor - \beta \cdot (u_i - l_i)\} \leq x_i\}$
  if P^{aux} *became infeasible* **then**
    Backtrack and add restriction $\{x_i \leq \max\{l_i, \lfloor \hat{x}_i \rfloor - \beta \cdot (u_i - l_i)\}\}$ to P^{aux}
  Add and propagate domain reduction $\{x_i \leq \min\{\lceil \hat{x}_i \rceil + \beta \cdot (u_i - l_i), u_i\}\}$
  if P^{aux} *became infeasible* **then**
    Backtrack and add restriction $\{x_i \geq \min\{\lceil \hat{x}_i \rceil + \beta \cdot (u_i - l_i), u_i\}\}$ to P^{aux}
/* The following loop is optional and forces solutions close to $\hat{x}$ */
for $i \in \mathcal{N}$ *with* $\hat{x}_i \neq \perp$ **do**
  if $l_i < u_i$ *in* P^{aux} **then**
    Add a continuous (nonnegative) variable s_i with objective coefficient $d_i = 1$ to P^{aux}
    Add a restriction $\{|\hat{x}_i - x_i| \leq s_i\}$ to P^{aux}
Solve P^{aux}
if P^{aux} *has feasible solution* x^{aux} **then**
  $x^{\text{compl}} \leftarrow x^{\text{aux}}$
else
  $x^{\text{compl}} \leftarrow$ **null**
return x^{compl}

costs in the optimal LP solution are fixed to their value in the LP solution. Additionally, all LP rows with nonzero dual multipliers are turned into equations and an equation $\{c^\top x = c^\top x^{\text{LP}}\}$ is added to the auxiliary problem to restrict the search space to the optimal LP face. Since the goal of this heuristic is to provide a feasible solution that lies on the optimal LP face (and hence terminate the solution process), it is only executed at dual-bound defining nodes. In addition to the usual working limits on the number of branch-and-bound nodes and minimum number of fixed integer variables in the auxiliary problem, this heuristic waits for a number of nonimproving branching decisions in the search tree before running, which is controlled by the parameter `heuristics/lpface/minpathlen`.

2.3.4 Locks heuristic

The locks heuristic is a LNS heuristic in the spirit of the clique and variable bound heuristics [62]. It is a pre-root heuristic that constructs its neighborhood based on structures in the problem, in this case based on the variable locks (see Definition 2.2.1). To do so, it iteratively fixes variables and performs domain propagation after each fixing in order to identify any resulting implications. After that, the resulting problem is solved as an LP and a simple rounding heuristic is applied on the LP solution. If the rounded solution is feasible, then a solution that is feasible for the original problem has been constructed, if not, the remaining problem is copied to P^{aux}, that is then solved with some working limits.

In the fixing step, the locks heuristic first sorts the binary variables based on the sum of each variable's down and up locks. Variables with the highest numbers of locks are treated first, since they are expected to be the most influential. Each variable in the sequence is then fixed to their upper (lower) bound if its number of up locks is smaller (higher) than its number of down locks. The motivation is that fixing the variable on this bound gives a higher chance to render constraints redundant and preserve feasibility of the remaining problem. If the number of locks is the same in both directions, variables are fixed randomly to one of their bounds, based on a given probability threshold. By default, a variable is rounded up with a probability of 67%, which proved to be reasonable in our computational experiments. After a variable was fixed and the change was propagated, the heuristic updates the locks of all variables. The update checks if any of the LP rows became redundant, based on their minimum and maximum activities after the last fixing. If a row is redundant, the variable locks for all variables in it are reduced.

Note that the constraint-based system of SCIP makes performing this update at the constraint level difficult. This is due to each constraint being treated as a black-box by the heuristic. Instead, the LP rows are used for this, which typically works well for MIPs, but might lead to incorrect updates of the locks for more general constraints. This is because one constraint may add multiple rows to the LP while locks are only added for the original constraint itself. While such incorrect updates may impact the fixing decision, the heuristic algorithm is still correct.

2.3.5 Computational Results

In order the assess the individual performance of the three heuristics LP face, locks and GINS, a computational experiment has been conducted using the MIPLIB 2010 benchmark test set [85] with a time limit of 2 hours. A configuration in which all three heuristics are activated serves as base line for comparisons to three configurations that disable one of the heuristics each.

GINS and LP face both incur a slight slow-down of 1.5%. GINS finds improving solutions for 15 of the 27 instances where it is executed, which shows the general purpose applicability of this heuristic. The structural requirements regarding the variable constraint graph are satisfied reasonably often. In contrast, LP face does not find any solutions on the MIPLIB 2010 benchmark set. However, in experiments on permuted instances (see Section 8), the heuristic is able to find

the optimal solution for 40 % of the permutations of instance `lectsched-4-obj` very fast, leading to significant performance improvements in these cases.

Finally, the locks heuristic has the most positive impact on the overall SCIP performance. Out of the 87 instances in the test bed, SCIP solves 2 more instances (72 instead of 70) and is 3.7 % faster if locks is active. The locks heuristic finds improving solutions on 41 instances. In the default settings, all three heuristics are enabled.

2.4 Reoptimization

Since version 3.2, SCIP provides *reoptimization* that can be used to warmstart the branch-and-bound tree after modifying the objective function or restricting the search space, see [64]. Consider a (finite) sequence of MIPs (P_i)

$$\min \{c_i^\top x \,:\, Ax \geq b,\ l \leq x \leq u,\ x \in \mathbb{Z}^p \times \mathbb{R}^{n-p}\}, \tag{8}$$

which differ in the objective function. While solving problem P_i all following problems P_j with $j > i$ are unknown. A possible application and one of the major motivations for providing this feature is column generation. In this case, the sequence of the MIPs is formed by the pricing problems that need to be solved at each node.

The main challenge in warmstarting the branch-and-bound tree is to guarantee optimality, because state-of-the-art MIP solvers extensively use dual information, e.g., variable bound reductions based on the current cutoff bound or strong branching. Whereas a reduction is called dual if it preserves, at a minimum, only one optimal solution. In contrast to that, a reduction is called primal if all feasible solutions will be preserved. Thus, deductions based on dual information are not necessarily valid after changing the objective function.

In SCIP 3.2, reoptimization could only be applied if (8) is a mixed-binary problem. Between two reoptimization solves, SCIP rebuilds parts of the solution space pruned due to dual reductions at each branch-and-bound node. This rebuilding process can be performed using a single linear constraint if the problem is mixed-binary, see [64]. In contrast to that, using this approach for a general MIP will either yield a constraint program (CP) or additional integer variables need to be introduced for the problem to remain linear.

Since the approach to rebuild parts of the search space pruned by dual reductions—which is described in the following—will enlarge the search tree, all methods using dual reductions except for strong branching will automatically be disabled when reoptimization is used. Therefore, we can restrict ourself to reduction performed on integer variables.

Let $\min \{c_i^\top x \,:\, Ax \geq b,\ l' \leq x \leq u',\ x \in \mathbb{Z}^p \times \mathbb{R}^{n-p}\}$ be a subproblem corresponding to a node of the branch-and-bound tree with local bounds $l \leq l' \leq u' \leq u$. Moreover, let $\mathcal{I}^\downarrow, \mathcal{I}^\uparrow \subseteq \mathcal{I}$ be disjoint subsets of integer variables such that for each x_i with $i \in \mathcal{I}^\downarrow$, the upper bound has been tightened to u'_i and for each x_j with $j \in \mathcal{I}^\uparrow$, the lower bound has been tightened to l'_j, both by dual deductions. In the case of a mixed-binary program, which was handled by SCIP 3.2, this is equivalent to fixing an integer variable to its lower or upper bound, respectively, i.e., $u'_i = 0$ for all $i \in \mathcal{I}^\downarrow$ and $l'_j = 1$ for all $j \in \mathcal{I}^\uparrow$. Thus, the part of the search space pruned by applying those reductions can be represented by creating a copy of the node without the dual reductions but extended by a single linear constraint

$$\sum_{i \in \mathcal{I}^\downarrow} x_i + \sum_{j \in \mathcal{I}^\uparrow} (1 - x_j) \geq 1. \tag{9}$$

In SCIP 4.0, the reoptimization feature is extended to be also applicable to general mixed-integer programs. If $\mathcal{I}^{\downarrow} \cup \mathcal{I}^{\uparrow}$ contain non-binary integer variables, a constraint of type (9) cannot be formulated. Thus, two possibilities are provided to reconstruct the part of the search space pruned by applying dual reductions. First, similar to (9), a so-called bound-disjunction constraint

$$\bigvee_{i \in \mathcal{I}^{\downarrow}} (x_i \geq u'_i + 1) \vee \bigvee_{j \in \mathcal{I}^{\uparrow}} (x_i \leq l'_i - 1) \tag{10}$$

is added to a copy of the node without applying the dual reductions. Second, SCIP 4.0 can perform interdiction branching [93] on $\mathcal{I}^{\downarrow} \cup \mathcal{I}^{\uparrow}$ w.r.t u' and l'. Therefore, $|\mathcal{I}^{\downarrow} \cup \mathcal{I}^{\uparrow}| - 1$ many nodes will be generated and a variable ordering on $\mathcal{I}^{\downarrow} \cup \mathcal{I}^{\uparrow}$ is needed.

2.5 Conflict and Dual Ray Analysis

In this section we give a brief summary of the results published in [139]. The analysis of infeasible subproblems plays an import role in solving MIPs. Most major MIP solvers implement one of two fundamentally different concepts to generate valid global constraints from infeasible subproblems. The first approach is called *conflict analysis* and has its origin in solving satisfiability problems and is similarly used in constraint programming, too. It is used to analyze a sequence of implications obtained by domain propagation that resulted in an infeasible subproblem. The result of the analysis is one or more sets of contradicting variable bounds from which conflict constraints can be generated. The second concept is called *dual ray analysis* and is based in LP theory and is used to analyze infeasible LP relaxations. The dual LP solution provides a set of multipliers that can be used to generate a single new globally valid linear constraint. In the following, we will focus on subproblems

$$\min\{c^{\top} x \,|\, Ax \geq b,\ l' \leq x \leq u',\ x_i \in \mathbb{Z}\ \forall i \in \mathcal{I}\} \tag{11}$$

with local bounds $l \leq l' \leq u' \leq u$ and an infeasible LP relaxation. In the following, the index set of all variables is denoted by $\mathcal{N}$ and the index set of all integer variables by $\mathcal{I} \subseteq \mathcal{N}$. The *dual LP* of the corresponding LP relaxation of (11) is given by

$$\max\{y^{\top} b + \underline{r}^{\top} l' - \overline{r}^{\top} u' \,|\, A^{\top} y + \underline{r} - \overline{r} = c,\ y, \underline{r}, \overline{r} \in \mathbb{R}^n_+\}, \tag{12}$$

By LP theory, each unbounded ray $(y, \underline{r}, \overline{r})$ of (12) proves infeasibility of (11). Moreover, the Lemma of Farkas states that exactly one of the following two systems is satisfiable

$$Ax \geq b, \quad l' \leq x \leq u' \tag{F_1}$$

$$y^{\top} A + \underline{r} + \overline{r} = 0, \quad y^{\top} b + \underline{r}^{\top} l' + \overline{r}^{\top} u' > 0, \quad y,\ \underline{r},\ \overline{r} \geq 0. \tag{F_2}$$

It follows immediately, that if (F_1) is infeasible, there exists an unbounded ray $(y, \underline{r}, \overline{r})$ of (12) satisfying (F_2). An infeasibility proof of (11) is given by the local bounds in combination with a single constraint

$$y^{\top} Ax \geq y^{\top} b, \tag{13}$$

which is an aggregation of all rows $A_{j\cdot}$ for $j \in \mathcal{M}$ with weight $y_j > 0$, where $\mathcal{M}$ is the index set of all model constraints.

In addition of using (13) for creating an "artificial" initial reason of the conflict graph (SCIP 3.2 and below), SCIP 4.0 uses this constraint directly for domain propagation in the remainder of the search. Since constraint (13) is a conical combination of model constraints, it is valid

within the global bounds. The infeasibility within the local bounds $[l', u']$ follows immediately from the Farkas Lemma, i.e., the *maximal activity*

$$\Delta(y^\top A, l', u') := \sum_{i \in \mathcal{N}:\ y^\top A_{\cdot i} > 0} (y^\top A_{\cdot i}) u'_i + \sum_{i \in \mathcal{N}: y^\top A_{\cdot i} < 0} (y^\top A_{\cdot i}) l'_i$$

of $y^\top Ax$ w.r.t. variable bounds $[l', u']$ is strictly less than the corresponding left-hand side $y^\top b$. This constraint along with an activity argument can be used to deduce local lower and upper variable bounds. Therefore, consider a subproblem with local bounds $[l', u']$. For any $i \in \mathcal{N}$ with a non-zero coefficient in the proof-constraint (13), the *maximal activity residual* is given by

$$\Delta^i(y^\top A, l', u') := \sum_{j \in \mathcal{N} \setminus i:\ y^\top A_{\cdot j} > 0} (y^\top A_{\cdot j}) u'_j + \sum_{j \in \mathcal{N} \setminus i:\ y^\top A_{\cdot j} < 0} (y^\top A_{\cdot j}) l'_j,$$

i.e., the maximal activity over all variables except x_i. Hence, valid local bounds are given by

$$\frac{y^\top b - \Delta^i(y^\top A, l', u')}{a_i} \left\{ \begin{array}{c} \leq \\ \geq \end{array} \right\} x_i \left\{ \begin{array}{l} \text{if } a_i > 0 \\ \text{if } a_i < 0 \end{array} \right. .$$

This procedure is called bound tightening [30], which is widely used in all major MIP solvers, for all kinds of linear constraints.

The analysis of dual rays is enabled by default in SCIP 4.0. In the following, we refer to this setting by `dualray`. We compared the performance of this new feature with a setting where the analysis of dual rays is disabled (`no-dualray`). For our computational experiments we used a collection of 254 instances taken from MIPLIB [23], MIPLIB 2003 [6], MIPLIB 2010 [85], Cor@l collection [92], ALU[1] test set, and markshare test set [40], that can be solved by `dualray`, `no-dualray`, or both within a time limit of 3600 seconds. Over the complete test set of 254 instances a reduction of solving time and generated branch-and-bound nodes by 5% and 7%, respectively, could be observed with the `dualray` setting. Moreover, use of dual ray analysis reduced the number of time outs from 12 to 8. On a subset of affected instances (105), where at least one setting analyzed more than 100 infeasible LP relaxations successfully, a reduction of time and nodes by 11% and 14%, respectively, could be observed. The `no-dualray` setting could not solve three of those instances within the time limit of 3600 seconds whereas all instances could be solved with the other setting. On a further subset of highly affected instances (22), where at least one setting analyzed more than 50.000 infeasible LP relaxations successfully, using `dualray` led to a reduction of time and nodes by 39% and 41%, respectively. No highly affected instances reached the time limit when dual ray analysis is enabled. In contrast, `no-dualray` could not solve two instances within the time limit.

2.6 Mixed integer nonlinear programming

A valuable feature of SCIP is the ability to solve mixed integer nonlinear programs. The current release of SCIP presents many extensions and improvements to the mixed integer nonlinear solver. The major improvements available in SCIP 4.0 include:

- A feature providing the KKT reformulation of mixed binary quadratic programs (Section 2.6.1),
- a multi-start primal heuristic (Section 2.6.2),

[1] The ALU instances are part of the contributed section of MIPLIB 2003.

- the OBBT propagator has been extended to solve nonlinear relaxations (Section 2.6.3), and
- a collection of new methods for generating outer approximation cuts for convex relaxations (Section 2.6.4).

2.6.1 KKT reformulation for mixed binary quadratic programs

SCIP 4.0 provides a new presolving method that improves the formulation of a (mixed binary) quadratic program with linear constraints by adding KKT-like conditions. This section presents a brief explanation of this method and computational results on randomly generated instances from the literature as well as instances from benchmark libraries.

Consider a *mixed binary quadratic program (MBQP)* of the form

$$\begin{aligned} \min \quad & \tfrac{1}{2}\, x^\top Q x + c^\top x \\ \text{s.t.} \quad & Ax \leq b, \\ & x_j \in \{0,1\} \qquad \forall\, j \in \mathcal{I}, \end{aligned} \tag{14}$$

where $Q \in \mathbb{R}^{n\times n}$ is symmetric, $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, and $\mathcal{I} \subseteq \{1,\dots,n\}$ is the set of integer variables. In the following, it is assumed that the feasible solution set S of (14) is nonempty and bounded. Note that the requirement of a symmetric matrix Q is not restrictive: In case that Q is not symmetric, it can be replaced by $\frac{1}{2}\,(Q+Q^\top)$ without changing the objective function.

If all variables are continuous, i.e., $\mathcal{I} = \emptyset$, then (14) is a *quadratic program (QP)*. Convex QPs with a positive semidefinite matrix Q and rational data can be solved in polynomial time via the ellipsoid method, see Kozlov et al. [86]. However, the method proposed in [86] does not cover the case of a nonconvex objective function. In the present work, the focus is on the case where MBQP can be nonconvex and $\mathcal{NP}$-hard, see Pardalos and Vavasis [108]. Nonconvex MBQPs have many applications, for instance, in production planning (see Baron [14]) and graph theory (see Motzkin and Straus [104]).

There also exist other solution approaches for (14) based on KKT reformulations: Chen and Burer [38] consider nonconvex QPs, and present an SDP-based branch-and-bound algorithm where the complementarity constraints of the KKT conditions are enforced by branching. Lin and Vandenbussche [91] investigate a KKT-like reformulation to solve QPs with fixed charge constraints of the form $0 \leq x \leq y$, where $x \in \mathbb{R}^n$ and $y \in \{0,1\}^n$.

In the following, a new reformulation for general MBQPs is proposed: Given $\tilde{x}_j \in \{0,1\}$ for $j \in \mathcal{I}$, consider the subproblem of (14) with the added constraints $x_j = \tilde{x}_j$ for all $j \in \mathcal{I}$. Since now all binary variables are fixed, the following KKT conditions are applicable:

$$\begin{aligned} Qx + c + A^\top \mu + I_{\mathcal{I}} \lambda &= 0, \\ Ax &\leq b, \\ x_j &= \tilde{x}_j \qquad \forall\, j \in \mathcal{I}, \\ \mu_i \cdot (Ax - b)_i &= 0 \qquad \forall\, i \in \{1,\dots,m\}, \\ \lambda &\in \mathbb{R}^{|\mathcal{I}|}, \\ \mu &\geq 0, \end{aligned} \tag{15}$$

where μ, λ are the Lagrangian multipliers of the linear constraints and $I_{\mathcal{I}}$ is the submatrix of the $n \times n$ identity matrix with columns indexed by $\mathcal{I}$. Introducing auxiliary variables $z_j \coloneqq x_j\, \lambda_j$

for all $j \in \mathcal{I}$, it can be shown for a KKT point (x, μ, λ) that

$$\begin{aligned} \tfrac{1}{2}\, x^\top Q x + c^\top x &= \tfrac{1}{2}\, x^\top (-c - A^\top \mu - I_{\mathcal{I}} \lambda) + c^\top x = \tfrac{1}{2}\, c^\top x - \tfrac{1}{2}\, x^\top A^\top \mu - \tfrac{1}{2}\, x^\top I_{\mathcal{I}} \lambda \\ &= \tfrac{1}{2}\, c^\top x - \tfrac{1}{2}\, \mu^\top b - \tfrac{1}{2} \sum_{j \in \mathcal{I}} x_j \lambda_j = \tfrac{1}{2}\, c^\top x - \tfrac{1}{2}\, \mu^\top b - \tfrac{1}{2}\, \mathbb{1}^\top z, \end{aligned} \tag{16}$$

where $\mathbb{1}$ is the vector of all ones of dimension $|\mathcal{I}|$. Observe that the constraints $z_j = x_j\, \lambda_j$ and $x_j \in \{0, 1\}$ imply the validity of the complementarity constraint $(1 - x_j) \cdot z_j = 0$. Therefore, $z_j = x_j\, z_j$ holds, and hence

$$z_j = x_j\, \lambda_j \quad \Longleftrightarrow \quad z_j\, x_j = x_j\, \lambda_j \quad \Longleftrightarrow \quad x_j \cdot (z_j - \lambda_j) = 0.$$

This proves that

$$\begin{aligned} \min \quad & \tfrac{1}{2}\, (c^\top x - b^\top \mu - \mathbb{1}^\top z) \\ \text{s.t.} \quad & (15), \\ & (1 - x_j) \cdot z_j = 0 && \forall\, j \in \mathcal{I}, \\ & x_j \cdot (z_j - \lambda_j) = 0 && \forall\, j \in \mathcal{I} \end{aligned}$$

is equivalent to the subproblem of (14) with $x_j = \tilde{x}_j$ for all $j \in \mathcal{I}$. Thus, the following *mixed binary program with complementarity constraints (MBPCC)* is equivalent to (14):

$$\begin{aligned} \min \quad & \tfrac{1}{2}\, (c^\top x - b^\top \mu - \mathbb{1}^\top z) \\ \text{s.t.} \quad & Qx + c + A^\top \mu + I_{\mathcal{I}} \lambda = 0, \\ & Ax \le b, \\ & x_j \in \{0, 1\} && \forall\, j \in \mathcal{I}, \\ & (1 - x_j) \cdot z_j = 0 && \forall\, j \in \mathcal{I}, \\ & x_j \cdot (z_j - \lambda_j) = 0 && \forall\, j \in \mathcal{I}, \\ & \mu_i \cdot (Ax - b)_i = 0 && \forall\, i \in \{1, \ldots, m\}, \\ & \mu \ge 0. \end{aligned} \tag{17}$$

Note that the objective function of (14) needs to be bounded on the feasible solution set S for this equivalence because the KKT conditions are only guaranteed to be satisfied for optimal solutions.

Feature description. SCIP 4.0 includes a reformulation method that automatically converts a given MBQP instance into an MBPCC instance. The complementarity constraints of (17) are handled by the SOS1 constraint handler in SCIP. If it is not explicitly specified in the settings, then the KKT reformulation is not performed for MBQP instances where the matrix Q is positive semidefinite or the feasible solution set S is not known to be bounded. By default, the KKT reformulation is only applied to QPs. For MBQPs with binary variables, the KKT reformulation should only be used for certain hard instances, as the computational results of the next section show.

To improve the performance of the branch-and-cut algorithm, it is often advantageous to keep the information about the quadratic objective function in the reformulated problem. This information can be extracted from the quadratic constraint $x^\top Q x + c^\top x + b^\top \mu + \mathbb{1}^\top z = 0$, which can be derived from (16). Adding this (redundant) constraint to (17) is useful for the generation of McCormick cuts [99] and can be exploited by propagation and by primal heuristics

(see Berthold et al. [20]). Nevertheless, in the default settings of SCIP, branching is exclusively performed on the complementarity constraints.

Moreover, with the help of an implication graph analysis (see [52]) the SOS1 constraint handler of SCIP tries to extract additional (redundant) complementarity constraints from the constraint system of (17). This additional data can be exploited during the whole solution process and may improve the branching rules and cutting planes that are implemented in the SOS1 constraint handler of SCIP. By default, implication graph analysis is turned off, but for certain problems it can be efficient as the following computational results show.

Computational results. The computational experiments were performed on a subset of the instance collection used in Chen and Burer [38] and instances from the libraries QPlib2014 [112] and MINLPLib2 [100]. The instances are of the following types:

- **BoxQP:** Box QP instances where $Ax \leq b$ coincides with $0 \leq x \leq \mathbb{1}$. The test set contains 20 of the 90 instances used in [38] that could not be solved in a few seconds, but in less than one hour by SCIP.
- **RandQP:** 20 of the 64 randomly generated QP instances generated by Chen and Burer [38] that could not be solved quickly.
- **LibsQP:** 83 instances taken from the benchmark libraries Globallib [73] and CUTEr [42]. The original test set used in [38] consists of 89 instances where 6 duplicates are removed here.
- **LibsMBQP:** 15 instances included from the libraries QPlib2014 [112] and MINLPLib2 [100]. The majority of these instances have a convex objective function.

The computational experiments were run with three different settings: one setting ("`KKTrefOff`") where the KKT reformulation is turned off and two settings ("`KKTrefOn`" and "`KKTrefOn-impl`") where the KKT reformulation is turned on. The nondefault settings of the latter two are:

- The KKT reformulation is also applied to convex MBQPs with unbounded variables.
- A higher priority was put for the enforcement of integer variables than for complementarity constraints; this means that the complementarity constraints were only enforced at search nodes with integral LP relaxation solution, which produced the best results on the LibsMBQP test set.
- Only for the setting `KKTrefOn-impl`, we additionally make use of an implication analysis to extract additional complementarity constraints from the constraint system.

Table 2 shows aggregated computational results on all four instance sets on a cluster of 64-bit Intel Xeon E5-2620 CPUs running at 2.10GHz. We used CPLEX 12.6.3 [81] as LP solver. Column "solved" lists the number of instances that could be solved within a time limit of one hour and column "time" the CPU time after the solving process terminated in shifted geometric mean with a shift of 10. The results show that `KKTrefOn` performs best for the BoxQP instances and that `KKTrefOn-impl` is the best choice for the RandQP and LibsQP instances among the three settings. For the LibsMBQP instances the best results regarding the CPU time were obtained if the KKT reformulation is turned off, but if the KKT reformulation is turned on, then two more instances can be solved within the time limit. One of these two instances has a nonconvex objective function. This indicates that for MBQP instances involving binary variables, the KKT reformulation might generally only be beneficial for hard instances. Surprisingly, for the BoxQP instances, the use of an implication graph analysis deteriorates the

Table 2: Computations with four test sets of (mixed binary) quadratic programs.

	BoxQP (20)		RandQP (20)		LibsQP (89)		LibsMBQP (15)	
Setting	solved	time	solved	time	solved	time	solved	time
`KKTrefOff`	0	3600.0	16	132.7	86	2.9	12	144.9
`KKTrefOn`	20	101.5	15	391.5	89	0.7	14	264.5
`KKTrefOn-impl`	12	667.0	19	53.5	89	0.6	14	244.2

solution time on the BoxQP instances. In additional tests, this was even true if all cutting planes for complementarity constrains were turned off. Future experiments on the KKT reformulation may involve other branching rules for complementarity constraints than the most-infeasible rule that is currently in use.

2.6.2 Multi-start Heuristic

The release of SCIP 4.0 contains a new primal heuristic based on a multi-start idea by Ugray et al. [135]. The heuristic applies multiple NLP local searches to a mixed-integer nonlinear program with possibly nonconvex constraints of the form $g_j(x) \leq 0$. The algorithm tries to identify clusters that approximate the boundary of the feasible set of the continuous relaxation by sampling and pushing randomly generated points towards the feasible region by using *consensus vectors* as introduced by Smith, Chinneck and Aitken [129]. For each cluster it uses a local search heuristic to find feasible solutions. This section presents a brief description of the heuristic and computational results on the MINLPLib2 benchmark library [100].

Algorithm. The multi-start heuristic first samples points in the domain $[l, u]$ and reduce the infeasibility of each point by using a gradient descent method. Afterwards, points that are relatively close to each other are grouped into clusters. Ideally, each cluster approximates the boundary of some connected component of the continuous relaxation of the MINLP. A reference point is computed by using a linear combination of all points of a cluster. This reference point is used as an initial start point for a local search heuristic. All steps of the heuristic are visualized in Figure 3. The following describes each of these steps in more detail.

sampling points First, K points $x^1, \ldots, x^K$ in the box $[l, u]$ are generated, where each point x^k is required to have a better objective value than the value $\mathcal{U}$ of the current incumbent solution, so $f(x^k) \leq \mathcal{U}$.

The domain of an unbounded variable x_i is reduced to a bounded box $[l_i, l_i + \alpha]$, $[u_i - \alpha, u_i]$, or $[-\frac{\alpha}{2}, \frac{\alpha}{2}]$ of size $\alpha \in \mathbb{R}_+$, depending on what variable bound is infinite. The default value is $\alpha = 10^4$. Integer variables are rounded to the closest integer value and will be considered in the following step as continuous variables.

reducing infeasibility For each point $\bar{x} \in \{x^1, \ldots, x^K\}$, a gradient-based method is used to reduce the maximum infeasibility $\max_{i \in \mathcal{M}} g_i(\bar{x})$, where $\mathcal{M}$ are the indices of constraints of the MINLP. To reduce the infeasibility of a point $\bar{x}$, the algorithm computes a descent direction

$$d_i := -\frac{g_i(\bar{x})}{||\nabla g_i(\bar{x})||^2} \nabla g_i(\bar{x})$$

for each violated constraint $g_i(\bar{x}) > 0$. Note that if $g_i(x) \leq 0$ is a linear constraint, the point $\bar{x} + d_i$ is the projection of $\bar{x}$ onto the hyperplane $g_i(x) = 0$. To update the current point $\bar{x}$, the constraint consensus method by Smith, Chinneck, and Aitken [129] is used. The idea is to consider a linear combination

$$\bar{x} \coloneqq \bar{x} + \sum_{i \in \mathcal{M}} \gamma_i \, d_i$$

with

$$\gamma_i \coloneqq \begin{cases} 0 & \text{if } g_i(\bar{x}) \leq 0, \\ |\{k \in \mathcal{M} : g_k(\bar{x}) > 0\}|^{-1} & \text{otherwise} \end{cases}$$

simultaneously. This is an alternative approach to considering each violated constraint individually like in alternating projection methods, which are explained in Boyd and Dattorro [29]. This update step is then iterated until $\bar{x}$ becomes feasible or a stopping criterion has been fulfilled.

clustering points All computed points of the previous step are grouped into clusters by using a greedy algorithm. The clusters (hopefully) approximate the boundary of the feasible set locally. The algorithm selects an unprocessed point and groups it with all points that have a small relative distance to it. The relative distance between two points x and $\tilde{x}$ is

$$\sum_{j \in \mathcal{N}} \frac{|x_j - \tilde{x}_j|}{|u_j - l_j|}.$$

Points with large violations are ignored. To reduce the computational effort, the maximum number of clusters is bounded, per default, by three.

solving sub-problems A local search heuristic is called for each identified cluster C. A starting point for the local search heuristic is computed by considering a linear combination of all points in C. Let

$$s \coloneqq \frac{1}{|C|} \sum_{x \in C} x$$

be the starting point for cluster C. In general, s does not satisfy the integrality condition of the MINLP. Before passing s to the local search heuristic, the multi-start heuristic rounds and fixes all integer variables to their closest integral value and solves the resulting NLP sub-problem.

Computational results. The computational experiments of this section were performed on the full MINLPLib2 benchmark library [100]. A time limit of one hour, a memory limit of 40 GB, and to avoid stalling, a gap limit of 10^{-4} are used. Each instance is solved with and without applying the multi-start heuristic. Due to the computational cost of this heuristic, it is called only once in the root node and given the lowest priority.

Aggregated results from the computational experiments are presented in Table 3. First, it can be observed that two more instances can be solved when using the heuristic. However, due to its computational costs, the heuristic slows down SCIP on easy instances, which are typically solved within seconds. This was the case on 9.3% of the instances, which results in an average

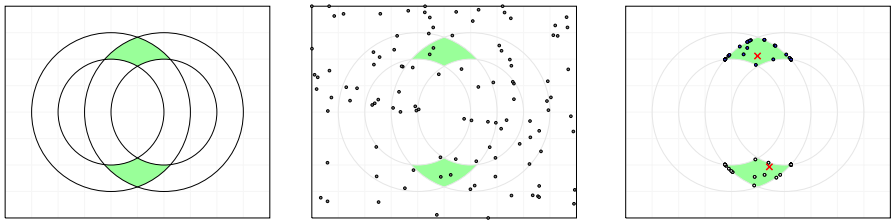


Figure 3: An example showing the different steps of the heuristic. The green colored area is the feasible region for the continuous relaxation of a nonconvex problem. The second picture shows the randomly generated points. After applying the gradient descent step and clustering the resulting points, the heuristic identifies two clusters. For each cluster it computes a starting point (marked with a red cross) by using a linear combination of all cluster points.

slow down of 3% on all instances that have been solved to optimality by both settings. The total number of branch-and-bound nodes does not change on average. On 469 purely continuous instances of MINLPLib2, an average speed-up of 2% and a node reduction of 2% can be observed when using the multi-start heuristic.

The heuristic found an improving, first, and best solution for 76, 15 and 34 instances, respectively. It finds an improving solution more often on NLPs than on MINLPs. This can be explained by the rounding step that is applied before calling the NLP solver. Rounding and fixing integer variables to their closest integer value will most-likely lead to infeasible NLP subproblems and thus no feasible solution can be found. For this reason, per default, the multi-start heuristic is only applied for continuous problems.

Table 3: Aggregated results for SCIP with and without applying the multi-start heuristic.

	MINLP (1367)			NLP (469)		
Setting	solved	time	nodes	solved	time	nodes
`default`	710	14.8	966	238	4.9	204
`multi-start`	712	14.8	964	238	4.8	200

2.6.3 Nonlinear Optimization-Based Bound Tightening

Optimization-based bound tightening (OBBT) is one of the most effective procedures to reduce variable domains of nonconvex mixed-integer nonlinear programs [113, 71]. It minimizes and maximizes variables over a convex relaxation of (1) to learn the best possible bounds with respect to this relaxation. In this section, a variant of OBBT using the convex nonlinear relaxation of the problem is presented. Optimizing over this set might lead to stronger tightenings of variable bounds compared to a linear relaxation. Additionally, this section presents a generalization of the so-called Lagrangian variable bounds (LVBs) by Gleixner and Weltge [72], which represent globally valid aggregations of all problem constraints. These aggregations can be used to derive fast additional bound tightenings during the full spatial branch-and-bound tree search.

First, the propagation algorithm is briefly described and it is shown how to use the dual solution of each sub-problem to derive LVBs. Finally, the impact of the algorithm is discussed on a large subset of instances from the MINLPLib2 [100].

Algorithm. In nonlinear optimization-based bound tightening (NLOBBT), auxiliary NLPs of the form

$$\begin{aligned} \min \,/\, \max \quad & x_i && && (18a)\\ \text{s.t.} \quad & g_j(x) \leq 0 && \forall j \in \mathcal{M}', && (18b)\\ & c^\top x \leq \mathcal{U}, && && (18c)\\ & -x \leq -l, && && (18d)\\ & x \leq u, && && (18e) \end{aligned}$$

are solved, where each g_j with $j \in \mathcal{M}' \subseteq \mathcal{M}$ is a convex differentiable function and $\mathcal{U}$ the solution value of the current incumbent. Note that (18) contains only the convex constraints of (1) and thus, the optimal objective value of (18) provides a valid lower/upper bound on variable x_i.

The propagator sorts all variables with respect to their occurrences in convex nonlinear constraints and sequentially solves convex NLPs of the form (18). Variables that could be successfully tightened by the propagator will be prioritized in the next call of the propagator on a new node in the branch-and-bound tree. By default, the propagator requires at least one nonconvex constraint in $\mathcal{M}$ to be executed. For purely convex problems, the benefit of having tighter bounds is negligible. Moreover, variables that do not appear in any of the nonlinear convex constraints will not be considered, even though they might lead to additional tightenings.

After solving an NLP to optimize x_i, it is possible to exploit the dual information to generate a globally valid inequality, called Lagrangian variable bound [71]. Let λ_j, μ, α, and β be the non-negative dual multipliers of the Constraints (18b), (18c), (18d), and (18e). Because of the convexity of g_j for each $j \in \mathcal{M}'$, it holds that

$$g_j(x) \geq g_j(x^*) + \nabla g_j(x^*)(x - x^*) \tag{19}$$

holds for every $x^* \in \mathbb{R}^n$. Let x^* be the optimal solution after solving (18) for the case of minimizing x_i (similar for the case of maximizing x_i). In the following, it is assumed that Slater's condition holds, i.e. $\exists x \in \mathbb{R}^n$ such that $g_j(x) < 0$ for all $j \in \mathcal{M}'$. Together with the convexity of (18) this implies that the KKT conditions

$$\begin{aligned} e_i + \lambda^\top \nabla g(x^*) + \mu c + \beta - \alpha = 0 \quad & && && (20a)\\ \lambda_j \, g_j(x^*) = 0 \quad & && \forall j \in \mathcal{M}' && (20b)\\ \mu c^\top x^* = \mu \mathcal{U} \quad & && && (20c)\\ \lambda_j \geq 0 \quad & && \forall j \in \mathcal{M}' && (20d)\\ \mu \geq 0 \quad & && && (20e) \end{aligned}$$

hold. Since the dual multipliers λ_j are non-negative, aggregating the inequalities $x_i \geq x_i$ and $\lambda_j \, g_j(x) \leq 0$ leads to

$$x_i \geq x_i + \sum_{j \in \mathcal{M}'} \lambda_j \, g_j(x), \tag{21}$$

which is redundant for the current bounds $[l, u]$, but possibly not for other locally valid bounds in a node of a branch-and-bound tree.

Instead of calling the, in general, very expensive NLOBBT propagator during the tree search, it is possible to use (21) to derive further reductions on x_i. By using the dual solution $(\lambda, \mu, \alpha, \beta)$ of (18) together with the KKT conditions (20), one can linearize (21). The resulting linear inequality, which proves that x_i is bounded from below by x_i^*, can be seen as a cheap approximation of NLOBBT in the branch-and-bound tree search. The details are as follows. First, $g_j(x)$ in (21)

is underestimated with (19) using the identities (20b) and (20c). Combined with the first KKT condition (20a) multiplied by $(x - x^*)$, one obtains a globally valid inequality

$$x_i \geq (\alpha - \beta - \mu c)^\top x + (\beta - \alpha + e_i)^\top x^* + \mu \mathcal{U}, \tag{22}$$

which can be seen as an underestimator for variable x_i.

Inequality (22) is called *Lagrangian variable bound*. SCIP handles such an inequality in the LVB propagator (implemented in `prop_genvbounds`), which propagates it in every node of the branch-and-bound tree.

An LVB is only added for x_i if $\alpha_i = \beta_i = 0$, otherwise the right-hand side of (22) depends on x_i itself and thus, most-likely, cannot be used to strengthen the bound of x_i.

Computational experiments. SCIP 4.0 has been extended by a new propagator plug-in `prop_nlobbt`. Since solving NLPs might be numerically difficult, the feasibility tolerance of the NLP solver is set to 10^{-8} (the SCIP default is 10^{-6}) to guarantee that the solution value x_i^* of (18) is indeed a valid bound for x_i with respect to SCIP's feasibility tolerances. Because of the potentially high computational cost, the propagator has the lowest priority, runs only in the root node of SCIP's branch-and-bound tree, and is deactivated by default. Inequalities (22), learned during propagation, are used in each node of the branch-and-bound tree whenever a new incumbent solution is found or a variable bound is tightened.

Experimental setup. For the conducted experiment, the overall performance impact on the complete branch-and-bound tree is analyzed. A comparison of the performance when enabling our new propagator plug-in to SCIP with default settings is shown. SCIP ran with a time limit of one hour per instance.

Test set. All instances of the MINLPLib2 benchmark library with at least one convex and one nonconvex nonlinear constraint after presolving have been selected. This set contains 318 instances, which are divided into two subsets. The first set, called ALLSET, contains all instances that have been successfully processed by both settings. The second set, called OPTSET, is a subset of ALLSET and contains all instances that have been solved to optimality by both settings.

Computational results. Table 4 contains all aggregated results comparing SCIP with and without NLOBBT on the instances of ALLSET and OPTSET. The columns show the number of solved instances, shifted geometric means of solving times and the number of processed nodes for each of the two subsets of instances.

By using NLOBBT, SCIP solves 5 more instances and processes the same number of nodes on OPTSET. On ALLSET a speed-up of 5% can be observed, which can be explained by the additional solved instances. Applying NLOBBT does not have a significant impact on solving time nor the number of nodes for instances in OPTSET.

Table 4: Aggregated results for SCIP with and without applying the NLOBBT propagator.

	ALLSET			OPTSET		
Setting	nsolve	nodes	time	nsolve	nodes	time
`SCIP`	156	6491	230.3	154	1179	17.1
`SCIP + NLOBBT`	161	5798	217.8	154	1169	17.5

2.6.4 Outer approximation cuts for convex relaxations

Two procedures for building tighter and/or deeper outer approximation cuts for quadratic convex constraints or, more general, convex relaxations are included in SCIP 4.0. The first, the *gauge separator*, is an extension of a method introduced in SCIP 3.2 to general convex relaxations. The second new separator is based on projecting onto a convex relaxation of the feasible region. Finally, a specialization of the previous method to convex quadratic constraints is presented.

Gauge function of a convex relaxation for separation. A technique for separating convex quadratic functions was introduced in SCIP 3.2, [63], Section 2.5. The basic idea of the technique is to build linearization cuts at a suitable boundary point of the relaxation represented by the quadratic constraint.

The motivation for such a procedure is that gradient cuts are, in general, not supporting for the feasible regions. This can easily be seen when considering the convex constraint $x^2 + y^2 \leq 1$ and separating the point $x = y = 1$. Indeed, the linearization cut at that point is given by $f(1,1) + \partial_x f(1,1)(x-1) + \partial_y f(1,1)(y-1) \leq 1$, that is $x + y \leq \frac{3}{2}$. Since the circle $x^2 + y^2 = 1$ does not intersect the line $x + y = \frac{3}{2}$, the linearization cut is not supporting.

A new separator plug-in `sepa_gauge` is implemented in SCIP 4.0, which computes linearization cuts that are supporting for more complex relaxations, such as, relaxations that contain general convex constraints. The following definitions are needed to provide an explanation of this method. Let

$$K = \{x : g_i(x) \leq 0 \, \forall \, i \in \mathcal{C}\}$$

be a convex relaxation of (1) where $\mathcal{C} \subseteq \mathcal{M}$ are the indices of the nonlinear convex constraints.

For now, assume that the origin is an interior point of K. The *gauge function* of K (see [118]) is

$$\varphi_K(x) = \inf\{t > 0 \colon x \in tK\}.$$

Let $\bar{x} \notin K$ be the point to be separated. The plug-in computes $\hat{x} = \frac{\bar{x}}{\varphi_K(\bar{x})}$, which is a point in the boundary of K, and then computes a linearization cut for any $g_i(x)$ that is active at $\hat{x}$. These linearization cuts are supporting for K.

However, there are two technical difficulties. The first difficulty is that 0 is not always an interior point of K. To handle this, the plug-in computes an interior point that is used to translate K to the origin. This computation is performed only once. To compute an interior point the problem

$$\begin{aligned} \min \quad & t \\ \text{s.t.} \quad & g_i(x) \leq t \quad \forall i \in \mathcal{C}, \\ & g_j(x) \leq 0 \quad \forall j \in \mathcal{L}, \\ & t \geq \underline{t}, \end{aligned}$$

is solved, where $\mathcal{L} \subseteq \mathcal{M}$ are the indices of the linear constraints and $\underline{t} \in \mathbb{R}_-$. The bound $t \geq \underline{t}$ is needed to bound the problem. The linear constraints are considered here so that the boundary point $\hat{x}$ satisfies the linear constraints whenever $\bar{x}$ satisfies them. Although there is no evidence that this is better in general, intuitively, an interior point to a tighter relaxation of the feasible region seems more appealing. If the plug-in does not find an interior point, it will be disabled.

The second difficulty is that, in contrast to the case when K is represented by a single convex quadratic constraint an explicit formula for the gauge function is not available. However, it can still be computed. Let x_0, not necessarily 0, be an interior point and $K' = K - x_0 = \{x$:

$g_i(x + x_0) \leq 0 \; \forall \, i \in \mathcal{C}\}$. Then $\frac{1}{\varphi_{K'}(\bar{x}-x_0)}$ is the solution of the one dimensional problem

$$G(x_0 + \lambda(\bar{x} - x_0)) = 0, \tag{23}$$

where $G(x) = \max_{i \in \mathcal{C}} g_i(x)$. Given that G is convex, the plug-in implements a binary search to solve (23). Note that in the case where $x_0 \neq 0$, $\hat{x} = x_0 + \frac{\bar{x}-x_0}{\varphi_{K'}(\bar{x}-x_0)}$.

The procedure reduces to computing a boundary point using an interior point plus a line search, an idea introduced by Veinott [137] and later applied by Kronqvist, Lundell and Westerlund in [87] in the context of extended cutting plane methods [138].

Projecting over a convex relaxation for separation. A natural method for computing a supporting hyperplane that separates an infeasible point from convex constraints is to project the point to be separated onto the boundary of the feasible region and create a gradient cut at the projection. A key difference between this method and the separation method presented above is that the projection does not need an interior point. On the other hand, more computational time is needed due to the fact that a convex NLP needs to be solved for every separation step, instead of computing an interior point once and perform a cheap line search for every separation.

The new separation plug-in `sepa_convexproj` computes the projection of the point to be separated, $\bar{x}$, onto a convex relaxation of the feasible region K, by solving $\min_{x \in K} ||x - \bar{x}||_2$. After computing the projection $\hat{x}$, it computes linearization cuts for any convex constraint that is active at $\hat{x}$. Since this separator is very computationally expensive, it is turned off by default.

Projecting over a single quadratic constraint for separation. The projection method described above can be implemented very efficiently when considering a single convex quadratic constraint. Let $f(x) = x^\top Ax + 2b^\top x$, where A is positive semidefinite. The projection problem for the set $S \coloneqq \{x \, : \, f(x) \leq c\}$ is

$$\min_{x \in S} ||x - \bar{x}||_2 = \min \Big\{ ||x - \bar{x}||_2^2 \, : \, x^\top Ax + 2b^\top x \leq c \Big\}. \tag{24}$$

Minimizing a quadratic objective over a quadratic constraint is a very well studied problem, and algorithms to find approximate global minimizers efficiently are well known [102]. However, it is shown here how to solve the problem for this concrete structure to make the computations needed for an implementation explicit. To this end, the interior of S is assumed to be non-empty.

To solve (24), first perform a change of variables to remove the bilinear terms. Let $PDP^\top$ be an eigenvalue decomposition of A and $y = P^\top x$. By substitution, (24) is equivalent to

$$\min \Big\{ ||Py - \bar{x}||_2^2 \, : \, y^\top Dy + 2b^\top Py \leq c \Big\}.$$

Since P is orthonormal, $||Py - \bar{x}||_2 = ||y - P^\top \bar{x}||_2$. With $\bar{y} = P^\top \bar{x}$ and $\bar{b} = P^\top b$, the problem can be rewritten as

$$\min \Big\{ ||y - \bar{y}||_2^2 \, : \, y^\top Dy + 2\bar{b}^\top y \leq c \Big\}.$$

From the optimality conditions (since $\bar{x} \notin S$), there exists ρ such that

$$2(y - \bar{y}) + (2Dy + 2\bar{b})\rho = 0 \quad \Leftrightarrow \quad (I + \rho D)y = \bar{y} - \rho\, \bar{b}.$$

Thus,

$$y_i(\rho) = \frac{\bar{y}_i - \rho\, \bar{b}_i}{1 + d_{ii}\, \rho},$$

and the problem reduces to finding ρ such that $\varphi(\rho) \coloneqq y(\rho)^\top D y(\rho) + 2\bar{b}^\top y(\rho) - c = 0$. It can be shown that $\varphi(\rho)$ is a strictly decreasing, strictly convex function that has a unique zero in $[0, \infty)$. For the existence of the zero, it is necessary that the interior of S is not empty. In this case, its unique root can be efficiently computed using Newton's algorithm.

Computational results. For the gauge separator, preliminary computational experiments show a slow-down of 25% on the convex instances of the MINLPLib2 [100]. When filtering out the easy instances, that is, instances that take less than 100 seconds with and without the separator, the slow-down is 10%. However, if the time spent in the separator is disregarded, there is a 6.8% speed up. A similar phenomenon occurs with `sepa_convexproj`. There is a general slow-down of 27%, but a speed-up of 5.6% when filtering out the easy instances and disregarding the time spent in the separator. These separators are turned off by default, due to this performance decrease. Nonetheless, the results show the potential of the methods and one focus of future development will be techniques for reducing the computational effort.

Regarding the projection on single quadratics, preliminary results show a speed-up of 2% on convex quadratic instances from MINLPLib2. Despite the improvement, more evidence is needed before this can be activated by default. In particular, no experiment showing the effect of these cuts on instances with convex and non-convex constraints have been performed. This will be investigated in the future.

2.7 Parallelization

Computational performance of a mathematical programming solver depends heavily on the speed of computer processors. With a greater focus on multi-core architectures, parallelization is a required development of mathematical programming solvers. In response to this demand, SCIP 4.0 contains the first internal shared memory parallelization framework for the solver. The internal parallelization framework, whereby the parallelization is implemented within the core of the code, complements the external parallelization already available with the UG framework, which is presented in Section 5.

The internal shared memory parallelization framework is a development project towards a full featured parallel mathematical programming solver. As part of this development, two different parallelization libraries—TinyCThread [134] and OpenMP [37]—have been used to implement the shared memory threading capability for SCIP, the details of which are provided in Section 2.7.1. As a first step in developing a parallel solver, a concurrent solver has been implemented in SCIP, which is now available in the current release. The concurrent solver is capable of using multiple mathematical programming solvers—either different solvers or the same solver with different configurations—in parallel for a single instance.

The availability of three different parallelization implementations in SCIP 4.0 presents an opportunity to evaluate any performance differences that may arise. Specifically, ConcurrentSCIP provides a shared memory parallelization that is completely deterministic, while FiberSCIP and ParaSCIP—as a part of the UG framework (Section 5)—provide an asynchronous communication scheme in shared and distributed memory environments respectively. As such, within this release the new parallelization feature of *distributed domain propagation* has been developed and the performance difference between ConcurrentSCIP and FiberSCIP is presented [76] (Section 2.7.3).

There are many challenges in the development of a deterministic parallel solver. The first is the definition of a clock, or counter, that provides a deterministic measure of time. Such a clock is used for the synchronization of data across different threads. The second major challenge is the development of an efficient method of communication between threads that introduces

little idle time across all threads. Finally, the sharing of information is necessary for a high performance parallel solver. Determining the type of information that should be shared and designing efficient transfer mechanisms are of high priority. The details of how these goals are achieved by the concurrent solver are described in the thesis of Gottwald [75]. A condensed version is provided in Section 2.7.2.

2.7.1 Shared memory parallelization interface

To be independent of the library used for parallelization, in SCIP 4.0 this functionality is wrapped into a seperate interface library. It provides a unified interface for locks and condition variables as well as functions to execute tasks in parallel. One implementation is based on TinyCThread [134] that exposes a platform independent API similar to the threading API in the C11 standard but compatible with older C standards. The parallelization interface then implements a simple thread pool with a work queue on top of it. A second variant uses OpenMP [37] to implement the interface. Because it makes use of the task directive it requires at least OpenMP version 3.0 and is therefore not compatible with Microsoft compilers. The parallelization library SCIP 4.0 uses can be selected with a new compile option. Supporting different options allows for some flexibility in the future development towards a fully parallelized solver.

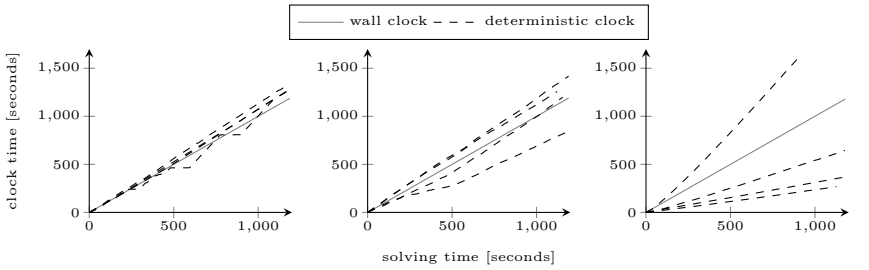
2.7.2 ConcurrentSCIP

In SCIP 4.0 several solvers can be executed concurrently to utilize multiple threads. When one of the solvers terminates, all others will be stopped. The approach was shown to be effective for small scale parallelization of MIP and SAT solvers. Fischetti et al. [55] have shown that in CPLEX the performance variability can be reduced and the performance increased by evaluating the root node with multiple solvers using different random seeds in parallel. Carvajal et al. [34] implemented this approach based on CPLEX and evaluated the impact of sharing different types of global information between the solvers, e.g. feasible solutions or cutting planes. They found the approach to be competitive to the default parallelization of CPLEX and the sharing of feasible solutions to be one of the best communication modes. For SAT solvers this approach is mostly called portfolio parallelization and has proven to be surprisingly efficient. It was first employed by the SAT solver ManySAT [78] which won the parallel track of the SAT competition in 2008. Since then it became the dominating approach for parallelizing state-of-the-art SAT solvers [10, 13, 12].

In ConcurrentSCIP the different solvers can use any settings and share feasible primal solutions as well as global variable bounds. Despite the communication of information between the threads, this new feature is deterministic—the solving process of all the solvers can be reproduced between multiple runs.

ConcurrentSCIP is implemented as a new plugin type called `concsolver`. An implementation of a concurrent solver needs to supply callbacks for setting up the problem, starting and stopping the solver as well as callbacks for communicating solutions and bounds. While this design allows different types of solvers to be used concurrently, SCIP 4.0 only includes an implementation for SCIP.

Deterministic clock. The instructions in a single-threaded program are always executed in the same order. In contrast, however, the order of instructions from multiple threads can vary non-deterministically. If the execution of a multi-threaded program shall be reproducible, the threads must share information at deterministic points in the program. Therefore in ConcurrentSCIP the solvers determine *communication points* using a deterministic clock [7] and do not exchange information between these points.



(a) instance biella1 **(b)** instance net12 **(c)** instance aflow40b

Figure 4: Deterministic clock with different settings

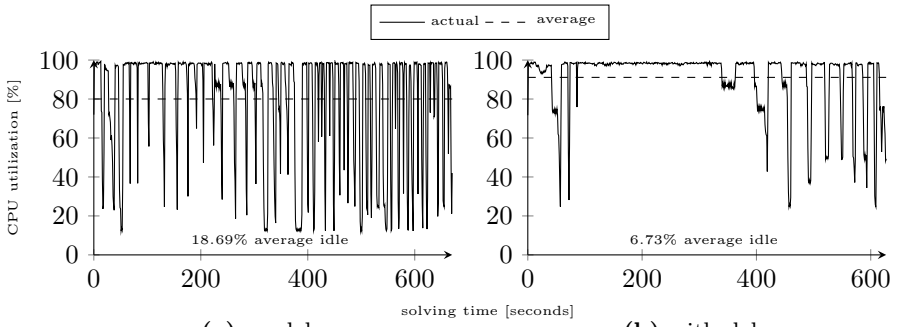
The deterministic clock in SCIP is implemented as a linear combination of statistics that are collected during the solving process, e.g. the number of LP iterations. To determine the coefficients of the linear combination a regression algorithm was used on data of the statistics and the corresponding wall clock time. This data was collected when running SCIP 4.0 with different emphasis settings on several instances of MIPLIB 2010 [85]. Because SCIP includes several heuristics that solve a sub-MIP it is important to make the statistics less dependent on the instance size. Therefore they were scaled with the number of non-zeros in the presolved problem.

For the regression we used an implementation of the *Lasso* [133] provided by scikit-learn [109]. This method applies a parameterized ℓ_1-regularization, which tends to yield a sparser solution and a smaller generalization error than ordinary least squares regression. Also other linear regression methods that use a different regularization were tested, but for the task at hand Lasso gave the best predictive accuracy. The parameter for choosing the amount of regularization was determined by cross-validation where the samples of one problem instance at a time were left out. The coefficients obtained were non-zero for the number of warm-started primal- and dual LP iterations, the number of bound changes in probing mode, and the number of calls to an internal function that checks whether solving should be stopped.

In Figure 4 the deterministic clock is compared to the wall clock when using different settings. Depending on the settings used, e.g. aggressive heuristics or separation, the deterministic clock behaves differently as the LP iterations within a heuristic and the LP iterations during the cutting plane separation are both only approximately converted to deterministic time. Still, in most cases the deterministic clock runs roughly at the same speed as the wall clock (Figure 4a and 4b), but on some instances it fails to give a comparable measure of time for different settings (Figure 4c). As a consequence some solvers have to wait for the other solvers regularly when they reach a communication point too early. Therefore it is important to design a deterministic communication scheme to cope with such behavior.

Synchronization. The deterministic clock can deviate between threads. This makes it difficult to maintain a high CPU utilization and a low communication overhead since solvers might need to wait for information shared by other solvers. In particular, a solver must wait if it wants to read information from a communication point that has not yet been reached by all solvers, otherwise this would incur non-determinism. If solvers are able to access shared information immediately, a *barrier* is required at each communication point, i.e., a point in the program when a thread is not allowed to continue if all other threads have not yet reached the barrier.

Because a barrier-based synchronization scheme can cause a large amount of idle time, the reading of data by solvers at a communication point is delayed by an amount d. If the deter-



(a) no delay **(b)** with delay

Figure 5: The CPU utilization of concurrent SCIP using 8 threads on the instance biella1, once without a delay and once using a delay.

ministic clock of a solver is at time t, it may only read data from communication points that occurred at time $t - d$ or earlier. Thereby the solvers are often able to exchange information deterministically without waiting at the expense of receiving slightly outdated information from other solvers. However, the positive impact of the delay on the CPU utilization (see Figure 5) outweighs this disadvantage, since solvers that are waiting frequently cannot contribute to the overall performance.

Communication points can occur when one of the statistics that are used in the deterministic clock are updated, i.e. when the deterministic clock advances. Then a synchronization event is generated if the elapsed deterministic time since the last communication point exceeds the current communication frequency. The solvers choose the delay and the initial frequency of communication heuristically based on the number of non-zeros and the number of variables. The frequency of communication is then adjusted dynamically based on the amount of the gap—difference between upper and lower bounds—that was closed between communication points.

2.7.3 Distributed domain propagation

Distributed domain propagation (DDP) exploits variable bound information in a concurrent solver to identify additional domain propagations. Due to the different settings that can be used with ConcurrentSCIP, the solvers will have different solution processes. Hence, the information used for domain propagation can be different between the solvers and bound reductions found in one solver may not be found in the other solvers. As such, DDP is able to perform additional domain reductions in each individual solver by sharing new global variable bounds. For details on the implementation we refer to Gottwald et al. [76].

2.7.4 Computational Results

Computational results for ConcurrentSCIP without DDP are provided in the thesis of Gottwald [75]. In summary, ConcurrentSCIP was shown to reduce performance variability and decrease the primal integral significantly. An evaluation for the impact of DDP in FiberSCIP and ConcurrentSCIP is provided in Section 5.1.1.

3 Interfaces

SCIP provides many interfaces to different programming languages. The most recent release delivers a new Java Interface (Section 3.1) and an interface to Julia (Section 3.2).

3.1 New Java Interface: JSCIPOpt

In SCIP 4.0, the already existing interface to the programming language Java has been vastly changed and is now called JSCIPOpt[2]. The interface is created via the wrapper and interface generator SWIG [131] and allows the user to model mixed-integer linear and quadratic programming problems.

So far, the Java interface has been created by parsing the documentation of SCIP. This approach is very error-prone and is not independent from the operating system. In contrast to this, SWIG automates this process and facilitates interface extensions. It automatically converts primitive data types from C to Java and generates for each C struct a corresponding Java class. The resulting interface is more robust and can be easily extended by customizing SWIG.

Furthermore, the new interface contains four simple basic Java classes to make it more user-friendly. `Scip.java`, `Constraint.java`, `Variable.java`, and `Solution.java` represent the structures `SCIP`, `SCIP_CONS`, `SCIP_VAR`, and `SCIP_SOL`. These Java classes implement the most important interface functions of `scip.h`, `pub_cons.h`, `pub_var.h`, and `pub_sol.h` in order to create and solve an optimization problem.

The following example creates and solves the MIP

$$\begin{aligned} \min \quad & x - 3y, \\ \text{s.t.} \quad & x + 2y \leq 10, \\ & x \in [2, 3], \\ & y \in \{0, 1\}, \end{aligned}$$

containing two variables and a linear constraint.

```
Scip scip = new Scip();
scip.create("MIP example");
Variable x = scip.createVar("x", 2.0, 3.0, 1.0,
   SCIP_Vartype.SCIP_VARTYPE_CONTINUOUS);
Variable y = scip.createVar("y", 0.0, 1.0, -3.0,
   SCIP_Vartype.SCIP_VARTYPE_BINARY);

Variable[] vars = {x, y};
double[] vals = {1.0, 2.0};
Constraint cons = scip.createConsLinear("linear", vars, vals,
   -scip.infinity(), 10.0);
scip.addCons(cons);
scip.releaseCons(cons);
scip.releaseVar(y);
scip.releaseVar(x);

scip.solve();
scip.free();
```

[2] https://github.com/SCIP-Interfaces/JSCIPOpt

```
using JuMP, SCIP

"Maximize c over integer points in the 2-ball of radius r."
function solve_ball(c, r, tol=1e-6)
    n = length(c)
    m = Model(solver=SCIPSolver())
    @variable(m, -r <= x[1:n] <= +r, Int)
    @objective(m, Max, dot(c,x))

    num_callbacks = 0
    function norm_callback(cb)
        num_callbacks += 1
        N = getvalue(x)[:]
        L = norm(N)

        if L > r + tol
            @lazyconstraint(cb, dot(N,x) <= r*L)
        end
    end
    addlazycallback(m, norm_callback)

    solve(m)
    return getvalue(x)[:], num_callbacks
end

sol, num_callbacks = solve_ball(rand(5), 50)
@show sol norm(sol) num_callbacks
```

Program 1: Short example on building a model with callbacks in JuMP and solving it with SCIP.jl. Adapted from http://www.juliaopt.org/notebooks/JuMP-LazyL2Ball.html

3.2 Julia Interface for JuMP: SCIP.jl

Julia, "a fresh approach to technical computing" [21], is a new programming language recently developed at MIT. Among other features, it offers concise syntax for array operations, good performance due to JIT compilation with type specialization via LLVM, powerful macro support, and the ability to call C functions directly.

Within the Julia community, there is a lot of interest in mathematical optimization, leading to the development of several packages by the *JuliaOpt* organization[3] [96]. In particular, the JuMP project provides an efficient, general modeling language, which can communicate with several mathematical programming solvers [47]. A key feature of JuMP is its support for the implementation of solver callbacks, such as lazy constraints, user cuts and primal heuristics. Importantly, the callbacks implemented in JuMP are completely independent of the underlying solver. Another Julia package, namely MathProgBase, defines an abstract solver interface, based on the common features of available solvers. JuMP only communicates to the solvers through MathProgBase.

With the SCIP.jl project[4], we wrap the SCIP solver for the Julia language and implement the necessary methods to make it available via MathProgBase. This enables users of JuMP to develop their model and callbacks once and then easily switch between solvers, such as Gurobi, GLPK and now also SCIP. The functionalities of SCIP currently available through SCIP.jl are: mixed integer programming, mixed integer nonlinear programming, lazy constraints, and primal heuristics. Note that SCIP.jl currently is the only solver interface to JuMP to support all of these features. However, SCIP.jl does not support obtaining duality information in linear or nonlinear programming.

A small example of SCIP.jl with JuMP using a callback is shown in program 1.

On a more technical level, the design of the MathProgBase interface was inspired by the common features of existing solvers. On the other hand, SCIP was designed as a framework extensible by plugins, with the goal of maximum flexibility. To bridge the significant conceptual gap between the two approaches we built a thin wrapper of SCIP with a limited feature set, simplified interface and some preset parameters, in the form of the C library CSIP[5].

4 SoPlex

Many performance improvements and features have been developed for the release of SoPlex 3.0. First, in an effort to reduce the numerical difficulties in SoPlex when used from within SCIP, the scaling capabilities have been improved. This involves preserving the scaled problem instance during modifications through the interface (see Section 4.1). Another new feature that is relevant with respect to SCIP is the *LP solution polishing* technique to find more integral bases (see Section 4.2).

As part of the current release, an experimental algorithm for a decomposition-based dual simplex has been implemented. This implementation aims to assess the potential of using decomposition within the simplex to address the negative effects of dual degeneracy. The details of the developed algorithm are presented in Section 4.3. This work is a dual counterpart of the decomposition-based algorithms developed to address primal degeneracy [50, 49, 48].

An unconventional feature of linear programming solvers that is available in SoPlex is the row representation of the basis matrix. This feature has existed since its initial development. However, there is limited information available regarding its use and implementation. Section 4.4

[3] http://www.juliaopt.org
[4] https://github.com/SCIP-Interfaces/SCIP.jl
[5] https://github.com/SCIP-Interfaces/CSIP

will provide a short description of the row form of the basis matrix, its use within SoPlex, and computational experiments regarding its performance impact.

4.1 Scaling

Scaling is a widely used means to improve numerical stability of linear programs [80, 110]. SoPlex has been no exception, applying equilibration scaling [110] by default and enabling the user to switch to alternative scaling methods. To further enhance numerical stability, the latest version of SoPlex incorporates additional scaling approaches that will be detailed in the following.

Broadly speaking, scaling of a linear program involves the multiplication of rows and columns by positive real numbers. For a mathematical view, consider a linear program in the following form

$$\min\{c^\top x : Ax = b,\ \ell \leq x \leq u\}. \tag{25}$$

Scaling of (25) can be described by means of two diagonal matrices $R = (r_{i,j})$ and $C = (c_{i,j})$ such that for the diagonal elements it holds that $r_{i,i} \in \mathbb{R}_{>0}$ and $c_{i,i} \in \mathbb{R}_{>0}$. The diagonals correspond to the row and column scaling factors respectively. Defining

$$A' = RAC, \quad b' = Rb, \quad c' = Cc, \quad \ell' = C^{-1}\ell, \text{ and } u' = C^{-1}u$$

one obtains the scaled linear program

$$\min\{c' \cdot x : A'x = b',\ \ell' \leq x \leq u'\}. \tag{26}$$

Each solution x' to (26) corresponds to a solution $x \coloneqq Cx'$ of (25) with the same objective value. With the expectation that (26) is more numerically stable than (25), the former will be solved in place of the latter.

It is important to note that in SoPlex scaling factors are always integral powers of 2. Consequently, scaling will only modify the exponent of the stored floating point, which guarantees that no roundoff error is added (apart from possible under- or overflow).

4.1.1 Least-squares scaling

A matrix is badly scaled if its non-zero entries are of vastly differing magnitude. While the opposite statement is not necessarily true, having non-zero entries of same or similar magnitude is generally seen as a desirable property (also beyond linear programming) and usually comes with improved stability.

The least-squares scaling approach suggested in [41] follows this precept and attempts to keep the absolute values of the non-zero entries of the constraint matrix close to 1. To this end, the algorithm first computes real numbers α_i and β_j for each row and column of the LP constraint matrix, respectively, such that the following expression is minimized:

$$\sum_{i,j:a_{ij}\neq 0} (\log_2 |a_{ij}| + \alpha_i + \beta_j)^2. \tag{27}$$

Next, for each i and j the closest integers $\tilde{\alpha}_i$ to α_i and $\tilde{\beta}_j$ to β_j are chosen, and the computed numbers are transformed to $r_{ii} \coloneqq 2^{\tilde{\alpha}_i}$ and $c_{jj} \coloneqq 2^{\tilde{\beta}_j}$. Setting the off-diagonal entries r_{ij} and c_{ij} to zero, one obtains scaling matrices R and C.

SoPlex 3.0 includes a least-squares scaling algorithm, as a tool to improve stability. To minimize (27) a specialized conjugate gradient algorithm is used, as suggested in [41]. Since least-squares scaling is computationally more expensive than equilibration scaling, it is not applied by default. It can be activated by the user with the command line parameter `-g5`. It is recommended that the least-squares scaling is only used for instances that are numerically difficult.

4.1.2 Persistent scaling in MIP solving

Besides being a stand-alone LP solver, SoPlex also takes a pivotal role within the SCIP Optimization Suite as an underlying LP solver for MIP solving. During branch-and-bound, numerically difficult LP relaxations may occur frequently even if the first root relaxation is well-behaved. Nevertheless, all previous releases of SoPlex only applied scaling to LPs if no warm start from an existing basis was performed (e.g. at the root node). In order to maintain the LP scaling throughout the entire branch-and-bound procedure—and allow for improved numerical stability—in SoPlex 3.0 *persistent scaling* has been implemented. This allows to consistently apply scaling during the MIP solving process by storing the scaling factors for the root LP and computing additional ones dynamically in case of newly inserted rows (e.g cutting planes) or columns.

To enable the persistent scaling approach, a new interface layer was added to SoPlex to ensure consistency (with respect to scaling) when external data is received or when internal data is accessed from outside of SoPlex. For example, bound changes found in SCIP need to be scaled before applying them in the LP and solution data of SoPlex (such as solution variables and reduced costs) must be unscaled accordingly. Operations involving the basis matrix of an LP, e.g., computing the i-th row of the basis inverse to generate Gomory mixed integer cutting planes, need to be transformed to represent the result with respect to the basis matrix of the unscaled LP.

Persistent scaling is now applied by default. Computational experiments with SCIP on MIPLIB 2010 [85] have shown no degradation in running time.

4.2 LP solution polishing

Solutions to (practical) linear programs are rarely unique. Instead, due to the presence of dual degeneracy (see also Section 4.3) and the use of numerical tolerances, multiple distinct solutions may fulfill the optimality and feasibility conditions. This is especially true for LP relaxations in MIP solvers and one of the main reasons for their performance variability [85].

Searching on the optimal facet for another LP solution is not a new idea, though, and has been investigated in many different ways. In Zanette et al. [145], the authors show how to exploit dual degeneracy by using the lexicographic simplex algorithm to find an optimal basis, which is then better suited to compute numerically stable cutting planes. A related approach for mixed integer programming is *k-sample* by Fischetti et al. [55]. This approach reruns the initial root LP several times on multiple cores using different random seeds. The aim is to collect different LP optima that provide richer cuts for the MIP solver. Alternatively, CPLEX [81] implements an algorithm [3] that fixes several variables and modifies the objective function to explore different optimal LP solutions to improve cut generation and to obtain more accurate dual information.

Algorithm. *LP solution polishing* tries to improve the quality of an existing LP optimum. The measure of solution quality in this case is the number of integer variables contained in the optimal basis. Since a non-basic variable is always on one of its bounds, it's also integral in the

current LP solution. Therefore, the less basic integer variables present in an LP solution the more integral it can be, which is usually a desired feature of relaxations in a MIP solver.

LP solution polishing, as described in Algorithm 5, starts after an optimal solution is found. A modified primal simplex is employed to avoid losing feasibility and the pricing step is modified to look for slack variables with zero dual multipliers to not deteriorate the objective function value. The modified ratio test only accepts the pivot to leave the basis if it is a problem variable. That way, in every successful iteration the number of problem variables in the basis can be reduced by one.

To avoid setting continuous variables to their bounds it is necessary to pass the integrality restrictions of variables in the MIP to SoPlex. The LP interface in SCIP 4.0 has been extended to facilitate this with negligible overhead, since the integrality information only needs to be updated after variables have been added to or removed from the problem. Currently, solution polishing is the only feature in SoPlex that makes use of this information.

In contrast to the method described in Achterberg [3], the presented algorithm does not modify the problem data and is not providing any intermediate information to the calling process—in this case SCIP. Furthermore, it is also possible to polish the solution of a pure linear program by treating all variables as integer variables. In this case a polished solution is considered better, because more variables are precisely on its bound rather than on some intermediate value in its feasibility range.

Algorithm 5: LP solution polishing of SoPlex

Input: Optimal (dual) solution x (y) with basis $\mathcal{B}$ of LP (25)
Output: Optimal solution of LP (25) with less or equal number of basic problem variables

set of problem variable indices $\mathcal{C} = \{1, \ldots, n\}$
set of slack variable indices $\mathcal{R} = \{1, \ldots, m\}$
set of non-basic indices $\mathcal{N} = (\mathcal{R} \cup \mathcal{C}) \backslash \mathcal{B}$
set of integer variable indices $\mathcal{I} \subseteq \mathcal{C}$
foreach $i \in \mathcal{N}$ **do**
    \\ find entering candidate among non-basic indices
    if $i \in \mathcal{C} \wedge i \in \mathcal{I}$ **then**
        \\ integer problem variable x_i is non-basic, hence on its bound
        continue
    else
        if $(c - A^\top y)_i = 0$ **then**
            \\ x_i has zero reduced cost (pivoting preserves optimal solution value)
            $j \leftarrow$ non-basic index in $\mathcal{B}$ chosen by primal ratio test
            if $j \in \mathcal{C} \wedge j \in \mathcal{I}$ **then**
                \\ found an integer problem variable x_j to leave the basis
                $\mathcal{B} \leftarrow \mathcal{B} \backslash \{j\} \cup \{i\}$ \\ perform basis change
                update x, y and $\mathcal{N}$
            else
                \\ no suitable index found to leave the basis, reject candidate i
                continue
return solution x, y and basis $\mathcal{B}$

Computational results. There are currently two different modes of using LP solution polishing within SCIP: running it only for the root LP solves and running it for every LP solve. In experiments on MIPLIB 2010 [85], the first setting of applying polishing only at the root node performed better. When compared to no LP solution polishing at all, the average number of nodes was reduced by 8% and the running time by about 10%, over the instances that are solved by both settings. However, simultaneously the number of instances solved within the time limit of two hours decreased by two when using LP solution polishing at the root. These results show the potential of solution polishing, but also the necessity to evaluate the benefit of polished LP solutions in more detail. Potentially it is better applied using a more fine-grained strategy. Hence, in the default settings of SCIP 4.0 LP solution polishing is deactivated.

4.3 Decomposition based dual simplex

Dual degeneracy is a phenomenon that is commonly observed in linear programs. A basis exhibits dual degeneracy when there exists at least one non-basic variable that has a zero reduced cost. Upon encountering a dual degenerate basis, the dual simplex method may perform a number of degenerate pivots. A degenerate pivot is characterized by the update of the basis without any improvement in the objective function value. Degeneracy can make the dual simplex method stall if a number of degenerate pivots are performed in succession. As such, dual degeneracy significantly impacts the effectiveness of the dual simplex method and can render highly degenerate LPs unsolvable.

Many attempts have been made to address degeneracy in the dual simplex method. The bound flipping ratio test [58] made a notable improvement to the performance of the dual simplex method in the presence of degeneracy. Alternatively, pricing rules have been developed by Forrest and Goldfarb [57] and Omer et al. [106] with the explicit intention to avoid degenerate pivots.

Degeneracy is a phenomenon that not only affects the dual simplex method, but also the primal simplex method. A basis is primal degenerate if there exists at least one basic variable taking a value of either its upper or lower bound. In an attempt to address the effects of primal degeneracy, which commonly arises in set partitioning problems, constraint aggregation approaches have been developed by Elhallaoui, Villeneuve et al. [50] and Elhallaoui, Metrane, Soumis and Desaulniers [49]. With a particular focus on column generation algorithms, constraint aggregation involved reformulating the restricted master problem to contain a subset of the original constraints, some of which represent multiple constraints from the original formulation. A dynamic procedure is employed that disaggregates the constraints, which is necessary to prove optimality. The reformulation approach presented by Elhallaoui, Villeneuve et al. [50] and Elhallaoui, Metrane, Soumis and Desaulniers [49] motivated the development of the decomposition-based *improved primal simplex* [48]. The improved primal simplex attempts to eliminate degeneracy by forming two partitions of the columns that currently form the basis—degenerate and non-degenerate—and using these partitions to perform a decomposition of the original problem.

While decomposition techniques have been shown to be successful in addressing primal degeneracy, very few investigations into their applicability for the dual simplex method have been performed. A fundamental part of the decomposition based approaches is the partitioning of the degenerate and non-degenerate variables. The pivoting rule of Omer et al. [106] describes a method to identify such a partition for a dual degenerate basis. This partitioning is then used to identify variables that when pivoted into the basis are expected to provide a strict improvement in the objective function value. Beyond the identification of degenerate and non-degenerate variables, there has been little work performed to develop a decomposition based dual simplex method.

The current release of SoPlex includes a development version of a decomposition based

dual simplex (DBDS) algorithm. This development version draws upon the concepts from the improved primal simplex in an attempt to reduce the impact of degeneracy using decomposition techniques. The implementation of the DBDS aims to investigate the potential of such a decomposition approach and identify possible future research directions.

4.3.1 Decomposition method

Consider the following linear program

$$\max\{c^\top x : Ax \leq b\} \tag{28}$$

with $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$. Without loss of generality, it is assumed that $m \geq n$. The dual of (28) reads

$$\min\{b^\top y : A^\top y = c,\, y \geq 0\}. \tag{29}$$

The dual simplex method can be applied to solve Problem (28). At each iteration of the dual simplex method a basis $\mathcal{B}$ is given. By definition, $\mathcal{B}$ is feasible for (29), but may be infeasible for (28). The basis $\mathcal{B}$ is given as the row-form, as described in Section 4.4—as such, $\mathcal{B}$ is of dimension $n \times n$. The primal-dual solution x, y corresponds to $\mathcal{B}$.

The partitioning of the rows in the original problem is based upon the values of y in the current basis $\mathcal{B}$. The inverse of the basis matrix $\mathcal{B}$ is denoted as Q, which will be used to perform the decomposition. The sets $\mathcal{P}$ and $\mathcal{N}$ describe this partition by containing the dual variables that have a non-zero and those that have a zero solution value, respectively. Given the partitioning of the dual variables $\mathcal{P}$ and $\mathcal{N}$, the concept of compatibility developed by Elhallaoui, Metrane, Desaulniers and Soumis [48] is used to decompose the original problem (28). Namely, the set $\mathcal{C}$ is defined to contain the indices of the rows i from the original problem that satisfy $A_i.Q_{.\mathcal{N}} = 0$, where $Q_{.\mathcal{N}}$ are the columns of Q indexed by $\mathcal{N}$. The rows satisfying $A_i.Q_{.\mathcal{N}} = 0$ are labeled the complementary rows. Note that the set $\mathcal{C}$ contains all rows contained in $\mathcal{P}$ and those that are not a linear combination of the rows contained in $\mathcal{N}$. For convenience, we define $\mathcal{I} := \{1, \ldots, m\}\backslash\mathcal{C}$ as the set of incompatible rows. Finally, the inverse of the basis can be reordered such that $Q = (Q_{.\mathcal{P}} \,|\, Q_{.\mathcal{N}})$.

By performing the substitution $x = Q\bar{x}$ and letting $x = Q_{.\mathcal{P}}\bar{x}_\mathcal{P} + Q_{.\mathcal{N}}\bar{x}_\mathcal{N}$, (28) becomes

$$\begin{aligned} \max \quad & (c^\top Q_{.\mathcal{P}})\bar{x}_\mathcal{P} + (c^\top Q_{.\mathcal{N}})\bar{x}_\mathcal{N} \\ \text{s.t.} \quad & A_\mathcal{C}.Q_{.\mathcal{P}}\bar{x}_\mathcal{P} + A_\mathcal{C}.Q_{.\mathcal{N}}\bar{x}_\mathcal{N} \leq b_\mathcal{C}, \\ & A_\mathcal{I}.Q_{.\mathcal{P}}\bar{x}_\mathcal{P} + A_\mathcal{I}.Q_{.\mathcal{N}}\bar{x}_\mathcal{N} \leq b_\mathcal{I}. \end{aligned} \tag{30}$$

By definition, $c^\top Q = y_\mathcal{B}^\top$, $c^\top Q_{.\mathcal{N}} = y_\mathcal{N}^\top = 0$ and $A_\mathcal{C}.Q_{.\mathcal{N}} = 0$. A decomposition can be performed by relaxing the incompatible constraints. This results in the following *reduced problem* for the DBDS:

$$\begin{aligned} \max \quad & (c^\top Q_{.\mathcal{P}})\bar{x}_\mathcal{P} \\ \text{s.t.} \quad & (A_\mathcal{C}.Q_{.\mathcal{P}})\bar{x}_\mathcal{P} \leq b_\mathcal{C}. \end{aligned} \tag{31}$$

In the DBDS, problem (31) is the master problem. The iterative algorithm of the DBDS augmented with additional rows throughout the solution process. The additional rows are sourced from the incompatible rows that were relaxed during the decomposition process.

In the DBDS, the reduced problem is solved to optimality to produce a basis that is used to identify whether incompatible rows can be added to continue the solution process. Such an optimal basis of (31) is denoted by $\mathcal{B}^*$, which has a corresponding dual solution that satisfies $y^*_\mathcal{C} \geq 0$. By setting $y^*_\mathcal{I} = 0$, the solution from (31) can be extended to a feasible dual solution of (28). Since (31) is solved to optimality, the dual solution vector y^* is expected to be different to

that corresponding to the basis $\mathcal{B}$. As a consequence, the sets $\mathcal{P}$ and $\mathcal{N}$, and respectively $\mathcal{C}$ and $\mathcal{I}$, must be redefined using the dual solution y^*.

The basis $\mathcal{B}^* \cup \mathcal{N}$, given by the solution to (31), is optimal for (28) if and only if there is a corresponding primal feasible solution. This is verified by solving the complementary problem, which is given by

$$\begin{aligned} \max \quad & s \\ \text{s.t.} \quad & A_{i\cdot}x = b_i \qquad \forall i \in \mathcal{P}, \\ & A_{i\cdot}x + s \leq b_i \qquad \forall i \in \mathcal{I}, \\ & x \in \mathbb{R}^n,\ s \in \mathbb{R}. \end{aligned} \tag{32}$$

If the objective function value of (32) is non-negative, then the basis $\mathcal{B}^*$ is optimal for the original problem. Otherwise, the solution given by (31) is infeasible for (28). The rows that are infeasible—those satisfying $A_{i\cdot}x^* > b_i$—can then be selected to augment the reduced problem (31).

Solving (32) can be viewed as an expensive pivoting rule for the dual simplex method. The hope from employing such a pivot selection rule is that the performance improvements gained from avoiding dual degeneracy is greater than the time required to set up and solve (32). As an alternative to solving (32), Omer et al. [106] propose a statistical approach for a pivot selection rule that identifies incompatible rows that are expected to provide non-degenerate pivots.

4.3.2 Implementation

The DBDS is implemented as an extension of SoPlex and builds upon the data structures provided within the linear programming solver. Central to the implementation of the DBDS is the row-form of the basis matrix that is employed by SoPlex. The use of the row-form is necessary in performing the required matrix multiplications to transform the variables and decomposition of the original problem.

A problem can be solved using the DBDS by setting the appropriate runtime parameters of SoPlex. The DBDS is activated by setting the parameter `bool:decompositiondualsimplex` to *true*. By activating the DBDS, the solution algorithm is automatically set to the *dual simplex algorithm*, the basis representation is set to *row* and persistent scaling is deactivated. Additional parameters are available to change different features of the DBDS algorithm. These parameters include `bool:usecompdual`, which informs the DBDS that the dual formulation of the complementary problem is solved; `bool:explicitviol`, if set to *true* will add all rows of the original problem violated by the reduced problem solution to the reduced problem; and `int:decomp_maxaddedrows`, which sets the maximum number of rows added to the reduced problem in each algorithm iteration. The solution algorithm is then performed in three main stages: initialization, decomposition, and termination.

Initialization. The decomposition approach of the DBDS requires a feasible dual basis to identify a partitioning of compatible and incompatible rows. The initialization phase starts by solving the original problem using the dual simplex method. Periodically during this initial solve the basis matrix is checked for the level of degeneracy, which is given by the number of non-basic rows with a zero reduced cost divided by total number of rows. Lower and upper bounds on the degeneracy level of 10% and 90% respectively are arbitrarily set within SoPlex as the required limits for executing the DBDS. The lower bound of 10% is imposed so that at least some degeneracy exists within the current basis. While the upper bound of 90% ensures that a few pivots are performed away from the 100% degenerate initial slack basis. In the case that the level of degeneracy does not fall between these two bounds after a predefined number of

checks, then the DBDS is abandoned and default SoPlex is called to solve the problem with the initial algorithm set to the dual simplex. The number of checks that are performed is given by a runtime parameter.

Decomposition. The decomposition of the original problem is performed using the degenerate basis found during the initialization phase. As a first step, the reduced problem is constructed by making the appropriate substitution of the variables and partitioning of the original problem rows. The identification of the compatible and incompatible rows is very computationally expensive and can be overly time consuming. Only half of the maximum allowed running time is provided to the decomposition phase. If this time is exceeded, then the decomposition approach is terminated and default SoPlex is invoked, with the initial algorithm set to the dual simplex, to continue solving the original problem.

The complementary problem is constructed after the first solve of the reduced problem. Starting from the original problem (28), the set of constraints that are currently tight in the optimal basis to (31) are set to equality constraints. A single variable is then added and is included in each of the incompatible rows. This additional variable is used to measure the constraint violations arising from the solution to (31). Finally, all rows that are not tight in the solution to (31) are removed. These operations result in a problem of the form given by (32).

The implementation of the DBDS allows the complementary problem to be solved as either the primal or dual formulation. As explained in Elhallaoui, Metrane, Desaulniers and Soumis [48], it is possible to eliminate many variables from the dual formulation of the complementary problem—expecting to improve computational performance. Currently the elimination of variables is not performed in the dual formulation. This is a point for future development.

The DBDS is an iterative algorithm that solves the reduced and complementary problem. Each solution to the reduced problem identifies the constraints in the complementary problem that must be set to equality. The solution to the complementary problem identifies whether the optimal basis of (31) is optimal for the original problem; if not, the violated rows that should be added to the reduced problem are given by the complementary problem solution.

Termination. The purpose of the termination phase is to resolve the original problem so that the solution from the reduced problem is transformed into the original problem space. At the successful completion of the decomposition phase, the optimal solution to (31) is relative to the substituted variables. To terminate the algorithm with the correct primal and dual solution vectors, a primal solution vector is setup by translating the solution from the reduced problem. The dual simplex algorithm is then executed to construct the dual solution vector and basis for the original problem.

Alternatively, the termination phase is required when it is not possible to execute the DBDS algorithm. This occurs if a dual degenerate basis is not discovered during the initialization phase, or the runtime was exceeded during the detection of the compatible and incompatible rows. In the first case, the default algorithm of SoPlex continues without interruption, but with the option to perform degeneracy checks disabled. The latter case invokes the default algorithm of SoPlex, with the initial algorithm set to the dual simplex method, after supplying the basis that was used to perform the decomposition of the original problem. Also, if an error occurs while executing the DBDS algorithm, then the same approach is used to resolve the original problem using the dual simplex method.

4.3.3 Investigation

The current implementation of the DBDS aims to investigate the potential of applying decomposition techniques to address the impact of degeneracy in the dual simplex method. The approach described above is very computationally expensive, hence it is not expected to be competitive with the default algorithm implemented in SoPlex with respect to time. As a measure of performance, the number of iterations performed in all solves of the reduced problem in the DBDS and the dual simplex iterations required to solve the original problem will be used for comparison. This analysis does not include the iterations of the complementary problem. It is deemed that solving the complementary problem is a pivot selection oracle, so the performed LP iterations are ignored in the analysis.

A set of instances has been collected from a wide range of publicly available test sets. An initial test set comprised 1235 LP instances collected from the following sources: COR@L[6], the Csaba Mészáros LP collection[7], the Hans Mittelmann benchmark instances[8], MIPLIB[9], and the Netlib LP test set including the "kennington" folder[10]. The DBDS algorithm was applied to all instances within this test set using a time limit of 1 hour. Since this is a development version of the DBDS, some instances abort due to insufficient memory. This occurred on 20 instances. These instances were the first removed from the test set. Next, the instances that could not be solved by the dual simplex method within 1 hour (50 instances) were removed. Then all other instance where the DBDS was executed and completed successfully were selected. This resulted in a test set consisting of 215 instances. This set of instances are those where a dual degenerate basis was found and the decomposition for the DBDS could be performed successfully.

Many different settings have been evaluated in the computational experiments of the DBDS. First, to provide an representative comparison, a 'default' SoPlex setting has been created. The default setting, labelled as `default_compare` sets the basis to use the row representation and the initial algorithm is set to the dual simplex. The DBDS is evaluated using four different settings. For all settings, the row representation is used for the basis matrix and the initial algorithm is set to the dual simplex method. Since the resolve of the reduced problem can be computationally expensive after adding rows, the impact of limiting the number of added rows is assessed. One setting permits the addition of at most 5 rows in each iteration, which has been arbitrarily chosen as a suitably small number of rows. The other setting adds all violated rows in each iteration. Finally, solving the complementary problem in the primal or dual form is also assessed.

Figure 6 presents a performance profile that compares the number of iterations performed by the default SoPlex implementation and the different settings for the DBDS. From this result, it is clear that the DBDS does not perform as well as default SoPlex on the collected test set. In particular, approximately 60% of all instances are solved by default SoPlex in the least number of iterations. Compared to all other settings, across all settings the DBDS performs the least number of iterations in 25% to 35% of instances.

This demonstrates that the decomposition based simplex is not valuable as a general algorithm and would be more suited as a problem specific implementation. From the current test set it is not clear what the best instances are for the DBDS. Additionally, it indicates that more research is required to identify whether there are benefits of such a decomposition approach.

[6] Computational Optimization Research At Lehigh. MIP Instances. http://coral.ie.lehigh.edu/data-sets/mixed-integer-instances/

[7] Csaba Mészáros. LP Test Set. http://www.sztaki.hu/~meszaros/public_ftp/lptestset/

[8] Hans Mittelmann. LP Test Set. http://plato.asu.edu/ftp/lptestset/

[9] Zuse Institute Berlin. MIPLIB–Mixed Integer Problem Library. http://miplib.zib.de/

[10] University of Tennessee Knoxville and Oak Ridge National Laboratory. Netlib LP Library. http://www.netlib.org/lp/

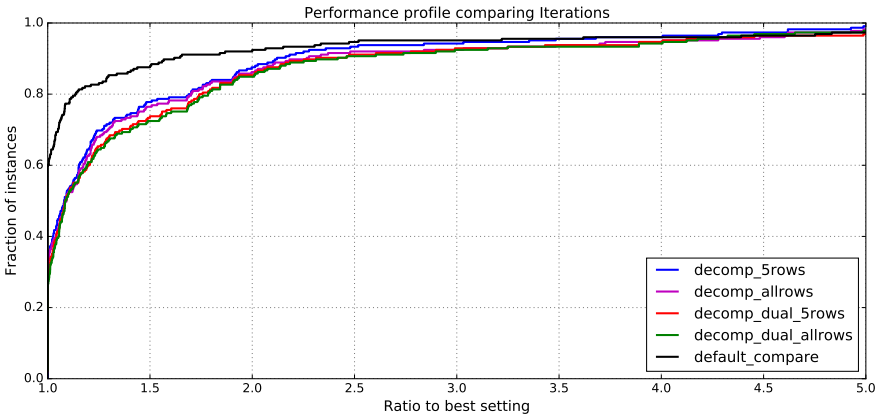


Figure 6: Performance profile comparing a default implementation of SoPlex against the decomposition based simplex when different numbers of rows are added in each algorithm iteration.

Investigations to identify the bottlenecks of the DBDS algorithm and identify an instance set where the algorithm would be most valuable are future directions of research.

4.4 Row basis

Two different representations of the basis are provided within SoPlex. The first is the commonly known column representation, where the basis is a square matrix with size equal to the number of rows. The other is the row representation, where the square basis matrix has a size equal to the number of columns. For this section, the following LP formulation is used:

$$\begin{aligned} \min \quad & c^\top x \\ \text{s.t.} \quad & Ax \geq b, \\ & x \geq 0. \end{aligned} \tag{P}$$

The concept of having two different representations for the simplex algorithm has first been investigated in detail in Wunderling [141] and is also explained in Gleixner [70]. Both representations are best illustrated using the following unifying definition of a basis.

Definition 4.4.1 (basis, basic solution)**.** *Let $\mathcal{C} \subseteq \{1,\dots,n\}$ and $\mathcal{R} \subseteq \{1,\dots,m\}$ be the index sets of variables and constraints of (P), respectively.*

1. *We call $(\mathcal{C},\mathcal{R})$ a* basis *of (P) if $|\mathcal{C}| + |\mathcal{R}| = m$. Variables and constraints with index in $\bar{\mathcal{C}} := \{1,\dots,n\} \setminus \mathcal{C}$ and $\bar{\mathcal{R}} := \{1,\dots,m\} \setminus \mathcal{R}$, respectively, are called* nonbasic*.*
2. *We call a basis $(\mathcal{C},\mathcal{R})$* regular *if the vectors $A_{\cdot j}$, $j \in \mathcal{C}$, and e_i, $i \in \mathcal{R}$, are linearly independent.*
3. *We call a primal-dual pair $(x,y) \in \mathbb{R}^n \times \mathbb{R}^m$ a* basic solution *of (P) if there exists a regular basis $(\mathcal{C},\mathcal{R})$ such that*

$$x_j = 0, \quad j \notin \mathcal{C}, \tag{33}$$
$$A_{i\cdot} x = b, \quad i \notin \mathcal{R}, \tag{34}$$
$$y^\top A_{\cdot j} = c_j^\top, \ j \in \mathcal{C}, \tag{35}$$
$$y_i = 0, \quad i \in \mathcal{R}. \tag{36}$$

4. *A primal solution x is called* primal feasible *if $Ax \geq b$, $x \geq 0$. A dual solution y with reduced costs $d = c - A^\top y$ is called* dual feasible *if*

$$d_j = 0 \ \vee \ (d_j \geq 0 \ \wedge \ x_j = 0) \quad \forall j \in \{1, \ldots, n\} \tag{37}$$

and

$$y_i = 0 \ \vee \ (y_i \leq 0 \ \wedge \ A_{i\cdot} x = b) \quad \forall i \in \{1, \ldots, m\}. \tag{38}$$

Dantzig [44] designed the simplex method for what is called the *column representation* of (P), which continues to be the basis for most state-of-the-art implementations. Here, slack variables $s \in \mathbb{R}^m$ are introduced in order to obtain equality constraints:

$$\begin{array}{ll} \min & c^\top x, \\ \text{s.t.} & Ax - s = 0, \\ & x \geq 0, \\ & s \geq b, \end{array}$$

Given a regular basis $\mathcal{B} = (\mathcal{C}, \mathcal{R})$, the variables x_j, $j \notin \mathcal{C}$, and s_i, $i \notin \mathcal{R}$, are set to their bounds as prescribed by (33) and (34). The m remaining columns $A_{\cdot j}$, $j \in \mathcal{C}$, and $-e_i$, $i \in \mathcal{R}$, form the (full rank) basis matrix $M = (A_{\cdot\mathcal{C}} \,|\, -I_{\mathcal{R}}) \in \mathbb{R}^{m \times m}$. The values of the basic solution can then be computed by solving two systems of linear equations:

$$M \begin{pmatrix} x_{\mathcal{C}} \\ s_{\mathcal{R}} \end{pmatrix} = b$$

and

$$M^\top y = \begin{pmatrix} c_{\mathcal{C}} \\ 0 \end{pmatrix}.$$

The *row representation* on the other hand is obtained by treating variable bounds as inequality constraints. Here, the basis matrix $N \in \mathbb{R}^{n \times n}$ consists of rows $e_j^\top$, $j \notin \mathcal{C}$ and $A_{i\cdot}$, $i \notin \mathcal{R}$. So the primal vector x can be computed by solving

$$Nx = \begin{pmatrix} 0 \\ b_{\bar{\mathcal{R}}} \end{pmatrix},$$

which is identical to (33) and (34). If $z \in \mathbb{R}^n$ denotes the dual multipliers associated with the bound constraints $x \geq 0$, then the dual vector y is computed solving

$$N^\top \begin{pmatrix} z_{\bar{\mathcal{C}}} \\ y_{\bar{\mathcal{R}}} \end{pmatrix} = c,$$

complemented with $y_i = 0$ for $i \in \mathcal{R}$.

Consequently, every basis $(\mathcal{C}, \mathcal{R})$ defines both a column and a row basis matrix. More importantly, a column basis matrix is regular if and only if the row basis matrix is regular:

Lemma 4.1. *Let $(\mathcal{C}, \mathcal{R})$ be a basis of (P), then the vectors $A_{\cdot j}$, $j \in \mathcal{C}$, and e_i, $i \in \mathcal{R}$, are linearly independent if and only if the vectors $A_{i\cdot}$, $i \notin \mathcal{R}$, and $e_j^\top$, $j \notin \mathcal{C}$, are linearly independent.*

Proof. After appropriate reordering of rows and columns the basis matrices of both representations yield the following:

$$\det(M) = \det \begin{pmatrix} A_{\mathcal{R}\mathcal{C}} & I_{\mathcal{R}} \\ A_{\bar{\mathcal{R}}\mathcal{C}} & 0 \end{pmatrix} = \det(A_{\bar{\mathcal{R}}\mathcal{C}}) \cdot \det(I) = \det \begin{pmatrix} 0 & I_{\bar{\mathcal{C}}} \\ A_{\bar{\mathcal{R}}\mathcal{C}} & A_{\bar{\mathcal{R}}\bar{\mathcal{C}}} \end{pmatrix} = \det(N)$$

□

Hence, for a regular basis, both types of basis matrices M and N as defined above are regular. Note that if the numbers of variables and constraints differ widely, so do the dimensions of M and N and subsequently the effort for computing and updating the basic solution values. The row orientation of the basis is computationally advantageous for instances that have significantly more constraints than variables since the basis matrices are smaller.

SCIP and SoPlex automatically switch to the row representation if there are at least 20% more constraints than variables. This threshold can be controlled with the parameter `rowrepswitch`.

Furthermore, branch-and-cut algorithms frequently add new constraints to the LP relaxation. In this case, the row representation is better suited than the column form, because an existing LU factorization of the basis matrix can be updated and reused in a hot start since its dimension remains unchanged. Conversely, the column representation can better handle newly added variables in a branch-and-price context. The appropriate representation is set for every new LP solve, so it could happen that after adding many cutting planes to the problem, the row representation might be preferred.

We use the MMM test set for computational experiments, see Section 2.1.4 for a definition of the test set. About 35% of the instances in the test set have at least 20% more constraints than variables after SCIP's presolving, thus triggering the row representation when SoPlex solves the LP relaxation. On this subset of affected instances, the row representation leads to a 20% reduction in the total solving time of SCIP, while the number of nodes remains almost unchanged. When regarding the entire test set this amounts to a speed-up of 7%.

5 UG

The *Ubiquity Generator* framework UG [128] is a generic framework to parallelize an existing state-of-the-art branch-and-bound based solver, which is referred to as the *base solver*, from "outside." UG is composed of a collection of base C++ classes, which define interfaces that can be customized for any base solver (MIP/MINLP solvers). These allow descriptions of subproblems and solutions to be translated into a solver-independent form. Additionally, there are base classes that define interfaces for different message-passing protocols. Implementations of different ramp-up strategies[11], a dynamic load balancing scheme, check-pointing and restarting mechanisms are available as a generic functionality. More details regarding these features that are provided by the UG framework are presented in the papers of Shinano et al. [128] and Shinano et al. [126]. The branch-and-bound tree is maintained as a collection of subtrees by the base solvers, while UG only extracts and manages a small number of subproblems from the base solvers for load balancing. Typically, these subproblems are represented by variable bound changes.

Using the UG framework, two types of external parallelization have been developed for SCIP and are included in the SCIP Optimization Suite. One is ParaSCIP [126, 127], which can run on distributed memory computing environments, the other is FiberSCIP [128], which can run on shared memory computing environments. ParaSCIP has been used to solve previously unsolved instances from MIPLIB 2003 and MIPLIB 2010. It produced 14 optimal solutions for these instances so far, using up to 80,000 cores, i.e., 80,000 MPI processes.

UG 0.8.3 presents two new features. First, distributed domain propagation is now employed during racing ramp-up (Section 5.1). Second, UG has been extended to base MIP solvers that use distributed memory parallelization themselves (Section 5.2).

[11] *ramp-up* is the process undertaken by a parallel solver from the start of computation until all processors have been provided work for the first time.

5.1 Distributed domain propagation

Distributed domain propagation (DDP) has been implemented within the UG framework to share variable bound changes during the solving process of FiberSCIP and ParaSCIP. The main details and concept of DDP have been presented in Section 2.7.3, with a focus on the shared memory parallelization of ConcurrentSCIP. In this section we describe the specific implementation details of DDP for the UG framework.

FiberSCIP provides the capability to share a wide range of information between solvers during the solution process. The most important piece of shared information is new incumbent solutions found by any of the solvers. Additionally, with the introduction of DDP, all global variable domain changes are shared.

The main difference between the DDP implementations between FiberSCIP and ConcurrentSCIP is the method of communication. In FiberSCIP, the LoadCoordinator provides the functionality for controlling the parallel branch-and-bound tree search. In particular, the LoadCoordinator manages the transferring of nodes to different solvers, load balancing and the sharing of global information. As such, the LoadCoordinator is critical in the implementation of DDP. The best incumbent solution and the tightest lower and upper bounds for each variable are stored within the LoadCoordinator. Every global bound tightening found in one of the solvers is sent to the LoadCoordinator first, which controls the distribution among the remaining solvers. When a new solution or bound is passed to the LoadCoordinator and is better than those currently stored, then this new piece of information is broadcasted to all solvers immediately. As such, each solver asynchronously communicates updated bounds to all solvers.

Apart from the communication system, the implementation of DDP is identical in ConcurrentSCIP and FiberSCIP. Specifically, the same propagation and event handler plugins are used for both implementations, with only slight modifications for the different interfaces. Given the same implementation of DDP, it is possible to evaluate and report of the difference in communications systems. Note that DDP can only be applied during racing ramp-up.

5.1.1 Results

The experiments for DDP evaluate the solving performance of ConcurrentSCIP and FiberSCIP. While DDP has also been implemented for ParaSCIP, the focus of this study has been restricted to shared memory parallel implementations. To make a direct comparison between ConcurrentSCIP and FiberSCIP, the latter is set to run using racing ramp-up only. In short, racing ramp-up solves the same instance on all solvers using different parameter settings with the limited communication of incumbent solutions and variable bound changes. Since the settings used within FiberSCIP during racing cannot be changed, ConcurrentSCIP is configured to use the same settings.

In Table 5 the nodes and solving time are given for the winning solver. They where aggregated with a shifted geometric mean where the time was shifted by 10 seconds and the nodes where shifted by 100. As can be seen in the table, FiberSCIP and ConcurrentSCIP with the wall clock required less nodes and the solving time is smaller when DDP is enabled. For ConcurrentSCIP with the deterministic clock the results are less conclusive. We suspect that the parameter settings for the deterministic synchronization need to be adjusted, since they have been tuned on intermediate development versions of SCIP 4.0.

Table 6 shows the shifted geometric mean of the number of additional domain reductions that were found via DDP in the winning solver. Clearly more domain reductions are found when more threads are used. The reason is that each solver generates different data that is then used for domain propagation, as explained in Section 2.7.3. The large difference between Fiber-

Table 5: Comparison of ConcurrentSCIP and FiberSCIP with the default settings of SCIP 4.0. Instances where no domain reduction was found via DDP and instances that where not solved by all settings where left out.

		with DDP		without DDP	
		Time	Nodes	Time	Nodes
Solver	Settings				
FiberSCIP	4 threads	119.9	5129.5	118.8	5217.5
	8 threads	112.9	4087.6	120.2	4406.2
	12 threads	121.5	4294.3	123.7	4286.3
ConcurrentSCIP	4 threads	172.2	5354.9	172.9	5512.8
	8 threads	179.4	4971.4	182.1	4821.3
	12 threads	202.8	4976.6	205.8	4543.8
ConcurrentSCIP (Wall clock)	4 threads	136.2	5243.8	143.9	5631.0
	8 threads	140.7	4527.8	145.0	4660.2
	12 threads	152.6	4557.7	155.8	4799.4
SCIP 4.0	default			148.2	8556.1

SCIP and ConcurrentSCIP indicates that the immediate asynchronous communication of the bound changes, as implemented in FiberSCIP, might have some advantages for DDP. However, in ConcurrentSCIP the delay of communication makes it more likely that a solver finds a tighter bound for the same variable before the other domain reduction is received. Additionally, FiberSCIP will share each subsequent domain reduction of one variable individually, whereas ConcurrentSCIP communicates for each variable only the best bound that any of the solvers found between two communication points.

5.1.2 Extension

The computational results for DDP presented in Sections 2.7.4 and 5.1.1 have focused on its use within concurrent solvers—namely ConcurrentSCIP and racing ramp-up of FiberSCIP. In addition to using DDP during racing ramp-up, FiberSCIP and ParaSCIP provide the capabilities to share the global variable bound information generated in the racing stage between solvers in all stages of the branch-and-bound algorithm. Given the low communication overhead of transferring variable bound changes, it is expected that this will provide a significant benefit to the solving process.

5.2 Capability to handle distributed base solvers

An extension of the UG framework is the ability to parallelize distributed memory base solvers. This is to allow a parallelized solver to be employed on large-scale distributed memory computing environments. For example, an LP or SDP solver that employs distributed memory parallelism can be integrated into a large-scale parallel branch-and-bound solver with the use of UG. To provide the capability, the UG framework was extended for this release.

PIPS-SBB [105] is an implementation of a general branch-and-bound algorithm for two-stage Stochastic Mixed-Integer Programs (SMIPs). It is the first to solve LP relaxations by using a distributed-memory simplex algorithm that leverages the structure of SMIPs. In a first implementation, the branch-and-bound tree search itself was not parallelized[12]. The development

[12]By the end of 2016, PIPS-SBB itself had an internal tree search parallelization

Table 6: Comparison of the number of domain reductions that where found via DDP in ConcurrentSCIP and FiberSCIP. The domain reductions on the subset of integer variables are given additionally in the second column.

Solver	Settings	#Dom. red.	#Int. dom. red.
ConcurrentSCIP	4 threads	15.3	7.6
	8 threads	17.6	7.9
	12 threads	21.9	8.8
ConcurrentSCIP (Wall clock)	4 threads	16.0	8.7
	8 threads	18.8	9.8
	12 threads	27.9	14.3
FiberSCIP	4 threads	89.9	42.8
	8 threads	130.7	55.9
	12 threads	147.9	60.5

of a tree parallilization version ug[PIPS-SBB, MPI] has been possible by using an enhancement to UG which allows it to distribute an MPI Communicator to each distributed memory solver. An MPI Communicator provides a separate communication space in MPI programs. With this extension, UG is now able to handle distributed MIP base solvers.

5.3 Parallelization of multi-threaded MIP solvers

UG can be used to parallelize multi-threaded MIP solvers. For example, UG 0.8.3 has been used to parallelize the commercial MIP solver Xpress. There are two versions that are realized by using this UG distribution, ParaXpress and FiberXpress [127]. Since Xpress is a multi-threaded solver by itself, FiberXpress provides two levels of multi-threading. It has been used to investigate the parallel performance of an external parallelization as provided by UG.

In previous experiments, ParaSCIP has been shown to effectively handle up to 80,000 MPI processes. In ParaXpress, each process can run with multi-threaded Xpress. Therefore, ParaXpress can potentially handle over a million cores to solve a single instance if the suitable hardware is available.

6 GCG

Many mathematical programs expose a model structure that can be exploited by decomposition/reformulation methods like Dantzig-Wolfe reformulation [45] or Benders decomposition [15]. The reformulation entails solving a different relaxation (hopefully stronger than the original linear relaxation), which also implies an additional algorithmic burden like column generation. GCG [67, 146] is able to handle Dantzig-Wolfe reformulations and Lagrangian decompositions for input linear or mixed-integer linear programs. The current release GCG 2.1.2 is a bugfix release.

Background. Considering the doubly-bordered block-diagonal coefficient matrix of a MIP with k *blocks* as

$$\begin{bmatrix} D^1 & & & & F^1 \\ & D^2 & & & F^2 \\ & & \ddots & & \vdots \\ & & & D^k & F^k \\ A^1 & A^2 & \cdots & A^k & G \end{bmatrix},$$

a classical Dantzig-Wolfe reformulation can be applied when the F^i, $i = 1, \ldots, k$, and G are zero. Otherwise, the corresponding *linking variables* can be duplicated for each of the (at most k) subsystems they appear in and a Lagrangian decomposition be applied. The identity of copies is then enforced via additional constraints in the master problem of the Dantzig-Wolfe reformulation.

This matrix/model structure can be made known to GCG by providing an additional file in `*.dec`, `*.blk`, or `*.ref` file formats (which are also used by similar frameworks like DIP [114]). Moreover, GCG tries to detect such a structure using several heuristics and partitioning algorithms working on different graph/hypergraph representations of the matrix [18]. There are several experimental further *detectors* for other matrix forms like staircase structure, which are disabled by default. When compiled with the external bliss [83] library, GCG tries to detect whether a set of blocks D^i are identical, and if so, eliminates symmetry by aggregating these. The structure detection loop currently undergoes a considerable extension and a full re-design to be shipped with the next major release.

After a structure is chosen, the original problem is reformulated by using Dantzig-Wolfe reformulation. GCG then runs a fully generic branch-price-and-cut algorithm on the reformulated problem, which is called *master problem*. The k subproblems, each having coefficient matrix D^i for $i = 1, \ldots, k$, can be solved as MIPs by SCIP or CPLEX, or by using a dedicated pricing problem solver. Since the subproblems are independent, they can not only be solved sequentially but also in parallel. GCG maintains both the original and the master problem, with both having their own branch-and-bound trees. Technically, the main solving loop works on the original problem, while the master problem is represented by a relaxator plugin. The trees are kept synchronous, such that there is always a one-to-one correspondence between nodes in the branch-and-bound tree of the original problem and the one of the master problem. Furthermore, there is a mapping between (in particular solutions to) both models. Among the most important ingredients are various branching rules (on original variables, Ryan-Foster branching for set partitioning models [119], and generic Vanderbeck's branching) [136], primal heuristics [94], cutting plane separators on original variables [95], dual variable smoothing, and a column pool.

7 Other extensions

Many applications and extensions of the SCIP Optimization Suite have been developed to solve various classes of mathematical programming problems. The current release presents updates to three main applications that are a focus of current development: SCIP-Jack [65], PolySCIP [27] and SCIP-SDP [61].

7.1 SCIP-Jack – Steiner tree problem and variants

The Steiner tree problem in graphs (STP) is one of the classical $\mathcal{NP}$-hard problems [84]. Given an undirected connected graph $G = (V, E)$, costs $c : E \to \mathbb{Q}_+$ and a set $T \subseteq V$ of *terminals*, the problem is to find a tree $S \subseteq G$ of minimum cost that contains T. Besides the (classical) STP,

numerous additional Steiner tree problem variants have established themselves in the literature, often propped up by various practical applications [65]. Against this backdrop, SCIP-Jack has been created as an exact framework (within SCIP) that can solve 12 Steiner tree problem variants in a unified approach [65]. A distinctive feature of SCIP-Jack is the transformation of all Steiner tree problem variants into a directed Steiner tree problem (also known as Steiner arborescence problem), which allows for a generic solving approach with a single branch-and-cut algorithm. Despite its versatility, SCIP-Jack is highly competitive with the best (problem-specific) solvers for several Steiner tree problem variants, see Gamrath et al. [65].

General Enhancements. SCIP-Jack 1.1 presents substantially improvements compared to its predecessor in the last SCIP release. Many of these improvements are documented in Gamrath et al. [65] and Rehfeldt et al. [116], but even compared to these articles SCIP-Jack has become significantly faster. In the following, several miscellaneous enhancements will be described that have not found their way into the two just mentioned articles; either because of the largely technical nature of the enhancements, or simply because they had not yet been implemented at that time. In addition to the major new components covered in the subsequent description, several new presolving techniques have been developed. For the latter enhancements the interested reader is referred to the two articles cited above.

The high level improvements include a change in the default propagator of SCIP-Jack (see Gamrath et al. [65]), which now additionally employs reduction techniques to fix variables (of the underlying IP formulation [65]) to zero. Whenever ten percent of all arcs have been newly fixed during the branch-and-cut procedure, the underlying directed graph D is (re-) transformed into a graph G for the respective Steiner tree problem variant. All edges (or arcs) in G that correspond to arcs that have been fixed to 0 in D are removed. Thereupon, the default reduction techniques of SCIP-Jack are used to further reduce G and the changes are retranslated into arc fixings in D. Furthermore, the separation algorithm of SCIP-Jack, which is based on the warm-start preflow-push algorithm described in Hao and Orlin [79], has been completely reimplemented. The new separation algorithm is for many instances more than ten times faster than the old one. Notably, the underlying preflow-push algorithm is now bolstered by a global relabeling and a gap relabeling heuristic, see Cherkassky and Goldberg [39].

On a lower level, many small implementation enhancements contribute to the broader picture. For instance, the dual ascent heuristic has been extended to combine the implementation described in Pajor et al. [107] with the guiding solution criterion suggested in Polzin [111]. Also, the breadth-first-search algorithm, which is a major run-time factor in the dual-ascent implementation suggested by Pajor et al. [107], has been reimplemented and no longer uses the default queue of SCIP, but an ad-hoc implemented alternative. The latter reimplementation reduces the overall run-time for several large-scale instances (with more than 100 000 edges) by more than 50 percent. Another example for a minor, but important, change can be found in the reduction history management of SCIP-Jack: Each edge—or arc, depending on the Steiner tree problem variant—is endowed with a linked list of ancestor edges, which allows to restore each solution in the reduced graph to a solution in the original one. When, in the course of the preprocessing, ancestor lists of two edges are joined, it needs to be checked that no two elements point to the same (ancestor) edge—otherwise lists might become prohibitively large. This check was formerly done by transforming one of the two lists to an array and using sorting combined with binary search. However, this procedure proved to be a considerable slow-down factor for several Steiner tree instances. Therefore, a hashing approach is now used that makes use of the new `SCIPallocCleanBufferArray` method, which provides an array initialized with 0, but also needs to contain only 0 elements when released. Furthermore, one can show that if two edges that have a common ancestor are merged, the new edge cannot be part of an optimal solution

and can therefore be discarded. This reduction method has also been added to SCIP-Jack 1.1.

New primal heuristics. SCIP-Jack 1.1 comes with three new primal heuristics. While they have already been used for the STP [111], recent work by the creators of SCIP-Jack on graph transformation and reduction techniques [115, 116] allows to extend the three heuristics for the first time beyond the STP. For instance, the heuristics are used in SCIP-Jack for the well-known prize-collecting Steiner tree problem [82].

The first heuristic is the *prune* algorithm, which builds upon bound-based reduction techniques [111, 116]. In its original version, the bound-based reductions eliminate a vertex (or edge) by proving that any solution that contains it has a higher objective value than a best known solution. In the prune heuristic, the upper bound originally defined by the best known solution is reduced such that in each iteration a certain proportion of edges and vertices is eliminated. Thereupon, all exact reduction methods are executed on the reduced graph, motivated by the assumption that the (possibly inexact) eliminations performed by the bound-based method will allow for further (exact) reductions. To avoid infeasibility, a guiding solution is initially computed by using a constructive heuristic [65] and the elimination of any of the vertices or edges of the guiding solution by the, inexact, bound-based method is prohibited. Within SCIP-Jack the heuristic is called whenever a new best solution has been found.

The *ascend-and-prune* heuristic makes use of the dual-ascent algorithm described in Wong [140], which provides a dual solution to a directed STP such that all terminals can be reached from the root by paths with reduced cost 0. The ascend-and-prune heuristic considers the graph constituted by the undirected edges corresponding to paths with reduced cost 0 from the root to all additional terminals. On this subgraph a solution is computed by first employing an (exact) reduction package and then using the prune heuristic. Within SCIP-Jack, ascend-and-prune is performed after each execution of dual-ascent, in particular prior to the initiation of the branch-and-cut procedure.

The last heuristic, *slack-and-prune*, is conceptually similar to the prune algorithm, but uses the information provided by dual ascent to compute the lower bounds associated with the vertices and edges. Since the dual-ascent algorithm has to be performed repeatedly, ascend-and-prune is used to compute upper bound during the execution of slack-and-prune. However, requiring multiple dual ascent runs, the heuristic is computationally expensive and is therefore only executed after the first LP solve at the root node and as part of the recombination heuristic originally described in Gamrath et al. [65].

Computational results. To demonstrate the performance of SCIP-Jack 1.1 in comparison to the six months older version (SCIP-Jack 1.0) described in Gamrath et al. [65], computational experiments on four benchmark test sets were performed. The covered variants are the Steiner tree problem in graphs (STP), the rectilinear minimum Steiner tree problem (RSMTP), the prize-collecting Steiner tree problem (PCSTP), and the hop-constrained directed Steiner tree problem (HCDSTP). More information on the test sets can be found in Gamrath et al. [65]. For the computations a cluster of Intel Xeon X5672 CPUs with 3.20 GHz and 48 GB RAM was used and CPLEX 12.6 was employed as the underlying LP solver. Moreover, the overall runtime for each instance was limited by two hours.

The results of the experiments are provided in Table 7. The table lists in columns one and two the test set and the problem variant. Furthermore, columns three and four show the number of solved instances and the shifted geometric mean (with shift 1) of the runtime on the test set for SCIP-Jack 1.0. The next two columns show the corresponding information for version 1.1. The last two columns provide the relative change in the number of solved instances and the average runtime.

Table 7: Computational comparison of SCIP-Jack with its predecessor.

		SCIP-Jack 1.0 [65]		SCIP-Jack 1.1		relative change [%]	
Test set	type	solved	∅ time [s]	solved	∅ time [s]	solved	∅ time
vienna-i-simple	STP	75	218.9	83	64.2	+10.6	−70.7
estein60	RSMTP	12	2672.5	12	2122.5	–	−20.5
PUCNU	PCSTP	10	56.3	11	38.0	+10	−32.5
gr14	HCDSTP	14	523.7	14	179.2	–	−65.8

The results show that SCIP-Jack 1.1 runs substantially faster on all test sets, most notably on vienna-i-simple and gr14. Moreover, nine more instances can now be solved to optimality within two hours. Also, although not visible in Table 7, the average gap for the unsolved instances is reduced by more than 50 percent with version 1.1. The enhanced performance of SCIP-Jack is a result of both the general improvements and the new heuristics, albeit the former can claim the larger share. In particular, the new implementation of the separation algorithm is responsible for most of the newly solved instances.

In summary, the results mark a substantial improvement of SCIP-Jack as compared to its predecessor described in Gamrath et al. [65], which was already shown to be highly competitive with other state-of-the-art Steiner tree problem solvers.

7.2 PolySCIP – Multi-criteria optimization

Multi-criteria optimization is concerned with optimizing several conflicting objectives at once. It can be considered as a generalization of single-objective optimization with numerous applications, e.g., in sustainable manufacturing [123] or in traffic and logistics [124].

PolySCIP 2.0 [27, 147] aims at solving problems of the form:

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

where $c_1, \dots, c_k \in \mathbb{R}^n$ with $k \geq 2$ are given linear objectives and $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ describe a finite set of linear constraints on the solution vector x, see Fig. 7. From here on, the feasible domain is denoted by $\mathcal{X}$ and the image in objective space by $\mathcal{Y} \coloneqq \{(c_1^\top x, \dots, c_k^\top x) : x \in \mathcal{X}\}$.

In contrast to the single-objective case, it is generally impossible to compute a single solution that optimizes all objectives simultaneously. A feasible solution $x^* \in \mathcal{X}$ is *efficient* if there is no $x \in \mathcal{X}$ such that $c_i^\top x \leq c_i^\top x^*$, for $i = 1, \dots, k$, with $c_j^\top x < c_j^\top x^*$ for at least one j. The corresponding image $(c_1^\top x^*, \dots, c_k^\top x^*) \in \mathcal{Y}$ of an efficient solution $x^* \in \mathcal{X}$ is called *non-dominated*. The challenge given a multi-objective problem lies then in computing all non-dominated points, see Fig. 8.

An efficient solution $x^* \in \mathcal{X}$ that is optimal for $\min_{x \in \mathcal{X}} \sum_{i=1}^k \lambda_i c_i^\top x$ for some $\lambda \in \mathbb{R}_+^k$ is called *supported* efficient solution and its corresponding image $(c_1^\top x^*, \dots, c_k^\top x^*) \in \mathcal{Y}$ is a supported non-dominated point, see Fig. 9.

The previous version of PolySCIP [63] allowed the user to compute the set of supported non-dominated extreme points of $conv(\mathcal{Y})$ by using a weight space partitioning approach [16, 130]. The bookkeeping of the weight space polyhedron was done via a graph data structure using the library LEMON [90].

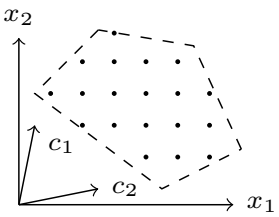


Figure 7: Bi-criteria integer programming problem with 2-dimensional feasible domain.

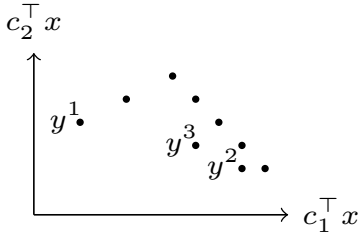


Figure 8: Image of a bi-criteria integer program with non-dominated points y^1, y^2, y^3.

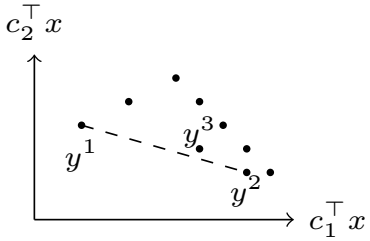


Figure 9: Image of a bi-criteria integer program with supported non-dominated points y^1 and y^2. The dashed line indicates that minimizing any convex combination of c_1 and c_2 yields y^1 or y^2, but not y^3.

New developments. A major revision of the source code was undertaken for the release of PolySCIP 2.0. Specifically, PolySCIP is now independent of any external library. Necessary computations with respect to the weight space polyhedron are now done via an implementation of the double description method [60]. Furthermore, version 2.0 offers the functionality to compute the entire set of non-dominated points (supported and unsupported) for integer problems with two or three objectives.

Bi-criteria case. The set of non-dominated points for bi-criteria integer programming problems is computed in the following way: Initially, both objectives are minimized independently, i.e., non-dominated extreme points y^i of $conv(\mathcal{Y})$, for $i = 1, 2$, are computed such that y^i minimizes c_i. If an *ideal* point $y \in \mathcal{Y}$ minimizing both objectives simultaneously is found, the problem instance is solved at this point. If the considered problem is unbounded for both objectives, corresponding unbounded rays are computed and the problem instance is considered to be solved at this point. If the problem is unbounded with respect to one of the objectives, a corresponding unbounded ray as well as a non-dominated extreme point of $conv(\mathcal{Y})$ with minimal value over all non-dominated extreme points of $conv(\mathcal{Y})$ with respect to the unbounded objective are computed, see Fig. 10.

Let $R(y^i, y^j)$ denote the rectangle in objective space given by $y^i, y^j \in \mathcal{Y}$ via the four points (y_1^i, y_2^i), (y_1^j, y_2^j), (y_1^i, y_2^j), (y_1^j, y_2^i). After *lexicographic* non-dominated extreme points y^1 and y^2 are found, the remaining non-dominated points that are located in the rectangle $R(y^1, y^2)$ are computed recursively. In a first step it is checked whether there is a feasible point $\bar{y}^3 \in \mathcal{Y} \setminus \{y^1, y^2\}$ located in $R(y^1, y^2)$. If there is no such feasible point $\bar{y}^3$, then there is also no further non-dominated point in the rectangle. If there is such a feasible point $\bar{y}^3$, then, in

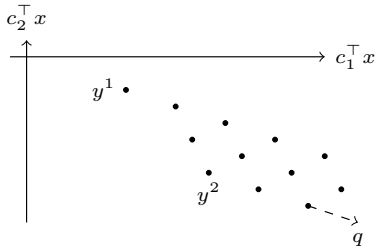


Figure 10: Part of an image of a bi-criteria integer program with infinitely many non-dominated points. y^2 is the non-dominated extreme point of $conv(\mathcal{Y})$ with minimal value for c_2. $y^2 + \ell q$, for $\ell \in \mathbb{N}$, generates an infinite sequence of non-dominated points.

a second step, a corresponding non-dominated point $y^3 \in \mathcal{Y} \setminus \{y^1, y^2\}$ located in $R(y^1, y^2)$ is computed. Subsequently, the rectangles $R(y^1, y^3)$ and $R(y^3, y^2)$ are checked for further non-dominated points. This procedure is repeated until all non-dominated points are computed.

To find out whether there is a feasible point (not necessarily non-dominated) in the rectangle $R(y^i, y^j)$ with y^i, y^j non-dominated and $y_1^i < y_1^j$, the following single-objective subproblem based on the weighted Tchebycheff norm [28] is solved:

$$\begin{aligned} \min \ & z \\ \text{s.t.} \ & -\lambda_1 r_1 \leq z - \lambda_1 c_1^\top x, \\ & -\lambda_2 r_2 \leq z - \lambda_2 c_2^\top x, \\ & y_1^i \leq c_1^\top x \leq y_1^j, \\ & y_2^j \leq c_2^\top x \leq y_2^i, \\ & x \in \mathcal{X}, \end{aligned} \tag{39}$$

where $r = (y_1^i - 1, y_2^j - 1)$ is a reference point and $\lambda = (1, \frac{y_1^j - r_1}{y_2^i - r_2})$ is chosen in a way that $x^i, x^j \in \mathcal{X}$ corresponding to y^i and y^j, respectively, yield the same objective value for (39) whereas any feasible solution corresponding to a point in the interior of $R(y^i, y^j)$ yields an objective value less than the one corresponding to x^i and x^j.

Let $\bar{y}^k \in \mathcal{Y} \setminus \{y^i, y^j\}$ be the feasible point located in $R(y^i, y^j)$ found after (39) was solved successfully, see Fig. 11. A corresponding non-dominated point y^k located in $R(y^i, y^j)$ will then be computed by solving:

$$\begin{aligned} \min \ & (c_1 + c_2)^\top x \\ \text{s.t.} \ & c_1^\top x \leq \bar{y}_1^k, \\ & c_2^\top x \leq \bar{y}_2^k, \\ & x \in \mathcal{X}. \end{aligned}$$

Tri-criteria case. For integer problems with three objectives a novel partitioning approach for the set of non-dominated points [122] is pursued. Let $\mathcal{N}_{-i}$, for $i = 1, 2, 3$, be the set of non-dominated points of $\mathcal{Y}$ whose corresponding efficient solutions are also efficient for the given integer problem where the i-th objective is being discarded. In a first step, a set of candidates $\mathcal{C}_i$, for $i = 1, 2, 3$, is computed by employing the bi-criteria solver capabilities. Then, for each

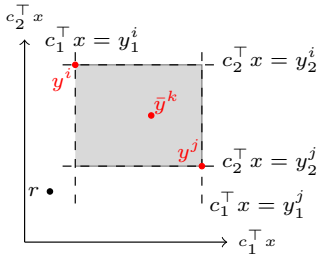


Figure 11: Feasible point $\bar{y}^k$ located in $R(y^i, y^j)$ that can be found by solving (39).

candidate $\bar{y} \in \mathcal{C}_i$, $i = 1, 2, 3$, a corresponding non-dominated point $y \in \mathcal{N}_{-i}$ is computed in a second step. In particular, let $\bar{y} \in \mathcal{C}_i$, for some $i \in \{1, 2, 3\}$, be a candidate point. By solving

$$\begin{aligned} \min \ & (c_1{+}c_2 + c_3)^\top x \\ \text{s.t. } & c_j^\top x = \bar{y}_j, \ \text{for } j \in \{1, 2, 3\} \setminus \{i\}, \\ & x \in \mathcal{X}, \end{aligned}$$

a corresponding non-dominated point $y \in \mathcal{N}_{-i}$ is computed.

The second partition, denoted by $\bar{\mathcal{N}}$, consists of all non-dominated points of $\mathcal{Y}$ whose corresponding efficient solutions are not efficient for any of the bi-criteria subproblems where one of the original objectives is discarded. The potential objective values of non-dominated points in $\bar{\mathcal{N}}$ are bounded from below and from above by points in the first partition as follows: Let $y \in \bar{\mathcal{N}}$ be a non-dominated point belonging to the second partition. Then there are $y^i \in \mathcal{N}_{-i}$, $i = 1, 2, 3$, such that

$$\begin{aligned} \max(y_1^2, y_1^3) \leq y_1 < y_1^1 \\ \max(y_2^1, y_2^3) \leq y_2 < y_2^2 \\ \max(y_3^1, y_3^2) \leq y_3 < y_3^3. \end{aligned}$$

In other words, the non-dominated points $y^i \in \mathcal{N}_{-i}$, $i = 1, 2, 3$, define rectangular boxes in objective space where non-dominated points belonging to $\bar{\mathcal{N}}$ might only be located. In PolySCIP 2.0, all possible boxes are created and made disjoint. Then, for each disjoint box $D := [b_1, e_1) \times [b_2, e_2) \times [b_3, e_3)$, the original tri-criteria integer program is restricted to D by adding constraints $b_i \leq c_i^\top x \leq e_i - \epsilon$, for $i \in \{1, 2, 3\}$ and some appropriate $\epsilon > 0$, and solved (recursively). Then, in a second step, each computed *locally* non-dominated point $\bar{y} \in D$ is checked for *global* non-dominance by solving the subproblem:

$$\begin{aligned} \min \ & (c_1{+}c_2 + c_3)^\top x \\ \text{s.t. } & c_i^\top x \leq \bar{y}_i, \ \text{for } i = 1, 2, 3, \\ & x \in \mathcal{X}. \end{aligned} \tag{40}$$

Let u the be optimal value of (40). $\bar{y} \in D$ is *globally* non-dominated if $u = \bar{y}_1 + \bar{y}_2 + \bar{y}_3$. Otherwise $\bar{y}$ is discarded as a dominated point.

Outlook. Solving multi-criteria optimization problems efficiently depends heavily on the number of subproblems that need to be solved to find new non-dominated points. For bi-objective and

tri-objective integer programs, respectively, the current implementation solves two subproblems, which is not ideal. The next version should decrease the number of subproblems that need to be solved. Furthermore, for tri-objective integer problems the number of *locally* computed non-dominated points, which are discarded as dominated, can become prohibitively large for larger instances. The next version should incorporate a way to compute only *globally* non-dominated points. Further research will focus on an extension and implementation towards integer problems with four objectives based on the partitioning approach described in [122].

7.3 SCIP-SDP

SCIP-SDP [149] is a plugin for mixed-integer semidefinite programming (MISDP) in SCIP. It can solve MISDPs using either a cutting plane approach, similar to how SCIP solves MINLPs, or a nonlinear branch-and-bound approach using interfaces to interior-point SDP-solvers. The newest release SCIP-SDP 3.0 adds an interface to the commercial SDP-solver MOSEK [103]. After a short introduction of SCIP-SDP, the implementation of the interface will be described before comparing its performance to the existing interfaces for DSDP [17] and SDPA [142, 143].

SCIP-SDP extends SCIP to mixed-integer semidefinite programs of the (dual) form

$$\begin{aligned} \inf \quad & b^\top y \\ \text{s.t.} \quad & \sum_{i=1}^{m} A_i\, y_i - C \succeq 0, \\ & \ell_i \leq y_i \leq u_i && \forall\, i = 1, \ldots, m, \\ & y_i \in \mathbb{Z} && \forall\, i \in \mathcal{I}, \end{aligned} \tag{41}$$

with symmetric matrices C, $A_i \in \mathbb{R}^{n \times n}$ for all $i = 1, \ldots, m$. It extends SCIP by a constraint handler and relaxator for SDPs, interfaces to multiple interior-point SDP-solvers, as well as two file readers (CBF and an extended version of the SDPA format) and multiple heuristics and propagators.

One of the main difficulties in solving general MISDPs lies in the non-inheritance of the Slater condition for the continuous relaxation of (41). The Slater condition, which requires the existence of a relatively interior point, is usually assumed for both primal and dual problem in the convergence theory of interior-point SDP-solvers, see for example Ye [144]. Therefore its failure may lead to numerical problems when solving the SDP relaxations. To handle these problems, SCIP-SDP uses a penalty formulation

$$\begin{aligned} \inf \quad & b^\top y + \Gamma r \\ \text{s.t.} \quad & \sum_{i=1}^{m} A_i\, y_i - C + I \cdot r \succeq 0, \\ & \ell_i \leq y_i \leq u_i && \forall\, i = 1, \ldots, m, \\ & r \geq 0, \\ & y_i \in \mathbb{Z} && \forall\, i \in \mathcal{I}, \end{aligned}$$

with identity matrix I and penalty parameter $\Gamma \geq 0$, to ensure the dual Slater condition. The same kind of penalty formulation is also used internally in DSDP. For a detailed description of SCIP-SDP and a discussion of the inheritance of the Slater condition in the branch-and-bound-tree see [61].

Table 8: Statistics of solver fails when the primal and dual Slater condition holds

solver	#relaxations	solved by default	solved by penalty	bounded by penalty	unsolved
DSDP	909 065	99.70 %	0.30 %	0.00 %	0.00 %
SDPA	757 899	90.12 %	6.53 %	0.00 %	3.35 %
MOSEK	1 551 194	99.58 %	0.42 %	0.00 %	0.00 %

Feature description. The newly interfaced SDP-solver of MOSEK implements an interior-point method based on the homogeneous self-dual embedding. It uses an extended formulation with auxiliary variables to detect ill-posed problems as well as infeasibility, which is a significant problem for other interior-point solvers.

Since MOSEK works on SDPs in primal form, instead of solving the relaxation of the dual formulation (41) the corresponding primal problem

$$\begin{aligned}
\sup \quad & C \bullet X - \sum_{i \in J_u} u_i v_i + \sum_{i \in J_\ell} \ell_i w_i & \\
\text{s.t.} \quad & A_i \bullet X - 1_{\{u_i < \infty\}} v_i + 1_{\{\ell_i > -\infty\}} w_i = b_i & \forall\, i = 1, \ldots, m \\
& X \succeq 0, & \\
& u_i \geq 0 & \forall\, i \in J_u, \\
& w_i \geq 0 & \forall\, i \in J_\ell
\end{aligned}$$

is given to MOSEK, where $J_\ell \coloneqq \{i \leq m : \ell_i > -\infty\}$ and $J_u \coloneqq \{i \leq m : u_i < \infty\}$ are the sets of finite variable bounds, 1_A is the indicator function of the set A and $X \bullet Y \coloneqq \operatorname{Tr}(XY)$ is the usual scalar product on the set of symmetric matrices. For the penalty formulation, the corresponding primal constraint $\operatorname{Tr}(X) \leq \Gamma$ is explicitly added to the problem.

Another difference between SCIP-SDP and MOSEK lies in the definition of feasibility tolerances. While SCIP-SDP uses an absolute tolerance for the smallest eigenvalue of the matrix, MOSEK uses a relative tolerance. Moreover, instead of taking the smallest eigenvalue of the matrix, MOSEK computes the largest absolute difference between $\sum_{i=1}^{m} A_i\, y_i - C$ and the current dual iterate, which always stays positive definite. To ensure feasibility for the absolute tolerance used in SCIP, the feasibility tolerance given to MOSEK is adjusted by dividing through $(1 + \max\{C_{ij}\})$ to revert the relative check in MOSEK. Furthermore, the default feasibility tolerance given to MOSEK is 10^{-7} in contrast to the 10^{-6} used in SCIP-SDP, since the check on the smallest eigenvalue is more restrictive than the check on the distance to the positive semidefinite cone. Computational experiments showed that this considerably reduces the originally high number of solutions not satisfying the feasibility tolerance in SCIP-SDP.

Computational results. The new MOSEK interface is compared to the existing DSDP and SDPA interfaces on the same test set used in [61], which has also become part of the conic benchmark library [59]. It consists of 194 instances in total, subdivided into 60 truss topology, 65 cardinality constrained least squares and 69 minimum k-partitioning instances, including some real world applications from cancer detection and very-large-scale integration (VLSI). The tests were carried out on a cluster of 64-bit Intel Xeon E5-2620 CPUs running at 2.10GHz using MOSEK 8.0.0.53, DSDP 5.8 and SDPA 7.4.0 together with preliminary developer versions of SCIP 4.0 and SCIP-SDP 3.0.

A comparison of the solver behavior dependent on the Slater condition is given in Tables 8–10. The tables show the total number of relaxations this state appeared in, the percentage of those relaxations that could be solved using the default formulation, the share of instances solved to

Table 9: Statistics of solver fails when either the primal or dual Slater condition does not hold

solver	#relaxations	solved by default	solved by penalty	bounded by penalty	unsolved
DSDP	45 781	99.83 %	0.11 %	0.00 %	0.04 %
SDPA	14 559	57.55 %	1.15 %	10.27 %	31.03 %
MOSEK	81 185	99.02 %	0.97 %	0.01 %	0.00 %

Table 10: Statistics of solver fails for infeasible subproblems

solver	#relaxations	solved by default	solved by penalty	bounded by penalty	unsolved
DSDP	83 522	91.98 %	2.23 %	1.53 %	4.26 %
SDPA	226 375	46.10 %	39.93 %	4.80 %	9.17 %
MOSEK	242 743	85.55 %	13.34 %	1.11 %	0.00 %

optimality or proven infeasible using the penalty formulation, the percentage of relaxations where a new lower bound could be computed using the penalty formulation and the share of nodes that could not be solved by either formulation. Note that these statistics are slightly biased towards the truss topology instances since the values represent averages over all relaxations and the truss topology instances tend to have the largest number of branch-and-bound nodes of all problem types in the test set.

It can be observed that MOSEK, like DSDP, rarely encounters any problems when the Slater condition holds. In case Slater's condition fails, especially for the truss topology instances, it can still solve over 99 % of all relaxations, while SDPA fails in more than 30 % of the cases. For DSDP the results are now similar to MOSEK, because, after further tuning of some parameters, two instances, which led to most of the unsuccessful solves in [61], can now be solved to optimality while avoiding those parts of the tree that failed the Slater condition. Good results can also be obtained with MOSEK for infeasible sub-problems, even though DSDP, which uses the same kind of penalty formulation as SCIP-SDP internally to detect infeasibility, performs even better now. Nevertheless, both can verify infeasibility for more than 85 % of the infeasible nodes, while SDPA, which only uses a generic primal-dual interior-point method without any specific techniques for detecting infeasibility, can only prove it for less than half of the instances.

The overall performance of the different interfaces is given in Table 11 and Figure 12. The number of branch-and-bound-nodes, the CPU time (with a limit of 3600 seconds) and the number of SDP iterations are given as shifted geometric means (with a shift of 100, 10 and 1000, respectively). The given percentages are the amount of SDP relaxations that were solved to optimality or proven infeasible using the penalty formulation and the share of relaxations, which could not be solved using any method. Both are given as arithmetic means over the percentages of each instance.

The MOSEK interface leads to an overall speed-up of more than a factor of two in comparison

Table 11: Solving times of SCIP-SDP for different SDP-solvers with default settings on test set of 194 instances

solver	solved	aborts	nodes	time	iters	penalty	unsucc
DSDP	174	0	390.45	158.19	11 923.20	0.20 %	0.41 %
SDPA	162	10	504.13	134.62	16 071.63	11.45 %	12.56 %
MOSEK	187	0	386.32	63.94	6527.21	2.95 %	0.12 %

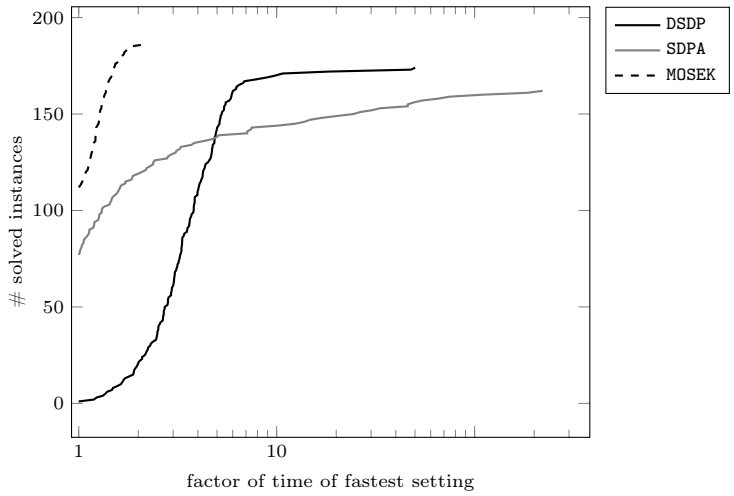


Figure 12: Performance plot for different solvers with best settings on test set of 194 instances

to SDPA and even more compared to DSDP, while also solving many more instances within one hour. Figure 12, however, shows that SDPA is still the fastest solver for almost half of the individual instances, but once it gets into numerical troubles, SDPA will often not be able to solve the instance at all, even given enough time. Moreover, while the difference between MOSEK and DSDP is consistently between a factor of two and three on each of the three instance sets, SDPA is slightly faster than MOSEK for cardinality constrained least squares, performs only slightly worse for truss topology, but is much slower for minimum k-partitioning.

8 Overall performance improvements

The development of the SCIP Optimization Suite is driven by two main factors. First, the addition of new functionality in order to increase the number and types of optimization problems that can be handled. Second, the improvement of solving performance. In the following we present the overall performance improvements achieved with the new features added for the SCIP Optimization Suite 4.0. We report both the number of solved instances and the shifted geometric means of the running time and of the number of branch-and-bound nodes, shifted by 10 seconds and 100 nodes, respectively. The results are presented with respect to MIP and MINLP separately. These two problem classes are a major focus of the development efforts for the SCIP Optimization Suite.

It is important to note that the results can only give a rough indication since they rely on benchmark sets of limited size. This is compensated for to some extent by generating for each instance, in addition to its original encoding, four equivalent versions by permuting variables and constraints. All five instances are theoretically equivalent. However, the phenomenon of performance variability, which is common to MIP and MINLP solvers, can lead to vastly different SCIP performance on each of the permuted instances. We aggregate, for each instance, the performance indicators as follows: for the number of solved instances we report the count of the instances solved in *all* permutations; for running times and nodes we compute the arithmetic average over all permutations of an instance before calculating the shifted geometric mean over all instances.

Alternatively, we could have measured performance variability by running SCIP 4.0 with different random seeds for the newly introduced parameter `randomseedshift` presented in Section 2.1.7. However, since this was not available in the previous release, we could not have used this to compare the performance of both versions.

The introduction of the random seed was chiefly motivated by the desire to reduce performance variability and a potential overtuning on standard benchmark sets. One expected consequence of the intensified randomization is of course an initial deterioration of the performance on the "default" permutation of the instances. Using intermediate development versions of SCIP before and after introduction of this feature, this deterioration was measured to be on average about 11% on MIPLIB 2010 [85], depending on the value for `randomseedshift`. When comparing running times over permutated instances, the deterioration was almost negligible (on average about 2%), in some sense validating the implementation of the randomization. Note that for the performance comparison on the "default" permutation the intensified randomization introduces a clear bias in favor of SCIP 3.2.1, but this effect is very much intended.

8.1 MIP performance

Table 12 compares the performance of SCIP 3.2.1 and SCIP 4.0 on the MIPLIB 2010 [85] benchmark test set, which contains 87 instances. Each job was run exclusively on a machine with Intel Xeon E5-2670 v2 CPUs with 2.50 GHz and 128 GB main memory, using a time limit of 7200 seconds and a memory limit of 35GB. The underlying LP solver was SoPlex—SoPlex 2.2.1 for SCIP 3.2.1 and SoPlex 3.0 for SCIP 4.0.

The results over the set of five permutations (including the default permutation) show that over all 87 instances SCIP 4.0 is faster by 7%. On the subset of instances that are solved within the time limit by both SCIP versions (on all permutations, columns under "OPT") the solving time is reduced by 8%. For this subset, the number of solved nodes is almost identical between the two versions.

When excluding the default permutation, the results show a stronger comparative performance by SCIP 4.0. The solving time is reduced by 9% on the complete test set and by 10% on the instances solved by both versions. It is also observed that the number of instances consistently solved within the time limit over all permutations, which is a more conservative measure than the average change in running time, increases from 65 to 66 instances.

In constrast, when solving only the default permutation of the instances the results are more mixed. First, SCIP 4.0 exhibits a slight deterioration in solving time by 1% on the complete test set and by 3% on the instances solved by both versions. This could be attributed to the intensified randomization and the reduction of overtuning. Second, the number of instances that SCIP 4.0 can solve within the time limit increases by 2. Finally, on the subset of harder instances—for which at least one SCIP version needs at least 600 seconds runtime—SCIP 4.0 is faster by 5%.

It is observed that SCIP 4.0 still exhibits the best performance on the default permutation of the instances. This is an effect also observed by other solvers. One possible explanation is that the default order of variables and constraints in the problem exhibits some structure. Specifically, related variables and constraints are often grouped together when generated by modeling tools. Some techniques inside MIP solvers may exploit this structure—leading to better performance. Since this may be a systematic benefit that is regularly encountered in real-world models, it is arguably better to measure performance and performance variability using multiple random seeds rather than multiple problem permutations. The changes to the handling of randomization, described in Section 2.1.7, now makes this possible when comparing future versions of SCIP.

Table 12: Aggregated results comparing SCIP 4.0 and SCIP 3.2.1 on the 87 instances of the MIPLIB 2010 benchmark set [85].

		ALL	OPT	
	solved	time	nodes	time
default				
SCIP 3.2.1	70	649.2	10892	351.7
SCIP 4.0	72	656.3	11795	362.3
permutations				
SCIP 3.2.1	65	785.5	14325	404.8
SCIP 4.0	65	732.7	14160	373.5
permutations without default				
SCIP 3.2.1	65	810.9	13974	430.4
SCIP 4.0	66	740.1	13376	387.8

8.2 MINLP performance

Table 13 compares the performance of SCIP 3.2.1 and SCIP 4.0 on 115 instances of the MINLPLib2 [100] benchmark library that are currently used by Mittelmann [101]. Each job was run exclusively on a cluster with a 64bit Intel Xeon X5672 CPUs at 3.20 GHz with 48 GB main memory. Each instance was solved using a time limit of one hour, a memory limit of 40GB, and a gap limit of 10^{-3}. For both versions the underlying LP solver was CPLEX 12.6.0.0 [81].

Compared to MIP, MINLP solving is in general numerically more difficult because nonlinear terms can more intensely magnify numerical errors. Therefore, Table 13 contains an additional column *failed* for the total number of instances for which SCIP terminates with a primal and dual bound that is inconsistent with the values stated at the MINLPLib2 website [100]. The value reported is the sum across all permutations.

The first, and arguably most important, observation is that the number of fails decreased significantly. This result indicates that the numerical stability of SCIP for MINLP has been greatly improved with the latest release. At the same time the number of solved instances decreased, which might partially be due to this increased numerical precision—less instances terminate prematurely. Note that SCIP 4.0 does not contain new features that systematically improve numerics, but several numerical bugs in the code were continuously corrected during the last development phase.

Similar to the MIP computational results, the average performance on the default permutation deteriorated. SCIP 4.0 processes 14% more nodes and needs 9% more time on the subset of instances that are solved by both SCIP versions (see columns under "OPT"). In contrast, the permutation runs show improvements in performance. On the four permutations excluding the default permutation SCIP 4.0 is on average 10% faster and needs 17% less nodes over the instances that are solved by both SCIP versions on all permutations. As for MIP, these result suggests that SCIP 3.2.1 was overtuned. More evidence on this conclusion can be taken from the average solution times over all instances. SCIP 3.2.1 experiences a greater slowdown when permuting the instances compared to SCIP 4.0.

8.3 Performance update for bugfix release 4.0.1

On September 1, 2017, the release of SCIP Optimization Suite 4.0.1 made the updated versions SCIP 4.0.1, SoPlex 3.0.1, and UG 0.8.4 available, which mostly contained bugfixes and minor

Table 13: Aggregated results comparing SCIP 4.0 and SCIP 3.2.1 on 115 instances of the MINLPLib2 benchmark library.

			ALL	OPT	
	solved	failed	time	nodes	time
default					
SCIP 3.2.1	63	3	903.3	66812	262.0
SCIP 4.0	56	1	1072.7	76305	286.6
permutations					
SCIP 3.2.1	53	10	1077.3	75835	306.5
SCIP 4.0	46	2	1144.5	66466	283.0
permutations without default					
SCIP 3.2.1	54	7	1076.8	70745	324.6
SCIP 4.0	49	1	1108.0	59108	293.8

interface updates. However, the fixing of some bugs also showed a slight positive effect on the performance of SCIP 4.0.1. On the default permutation we measured the following changes:

- On the MIPLIB 2010 benchmark set, SCIP 4.0.1 is on average 8% faster, although it solves two instances less within the time limit of two hours. On the instances solved both by SCIP 4.0.0 and 4.0.1, the runtime and node reductions are 10% and 15%, respectively.
- On the 115 instances of MINLPLib2 introduced above, SCIP 4.0.1 solves one more instance within the time limit of one hour and exhibits one less numerical fail. On the instances solved both by SCIP 4.0.0 and 4.0.1, the runtime and number of nodes are reduced by 15% and 18%, respectively.

9 Final remarks

This report has provided a view into the development of the SCIP Optimization Suite. There are many aspects that must be considered in the development of a solver, ranging from technical algorithm implementations to the theoretical mathematics underlying solver features. Each of the large range of topics may contribute to the solver performance. Neglecting technical algorithm implementations in favor of new theoretical developments may result in a theoretically advanced solver, but many challenges would remain open when striving for state-of-the-art solving performance. Similarly, focusing purely on technical aspects of a solver will overlook the cutting edge of mathematical optimization. As such, the paper has tried to touch on the breadth of research and development for the SCIP Optimization Suite. We hope that the detailed description provided in this report will bring awareness of the intricacies of developing mathematical programming solvers in general.

Acknowledgements

The authors want to thank all contributors to the SCIP Optimization Suite. Special thanks go to Roland Wunderling, the creator of SoPlex, Tobias Achterberg, the creator of SCIP, the former developers of SCIP, Timo Berthold, Stefan Heinz, Michael Winkler, and Kati Wolter, and the former GCG developer Martin Bergner. We are grateful for the support of Alexander Martin

who developed SIP, the predecessor of SCIP, and who fostered many of the latest developments in presolving.

The work for this article has been partly conducted within the *Research Campus MODAL* funded by the German Federal Ministry of Education and Research (fund number 05M14ZAM). The third and twelfth author would like to thank the German Research Foundation (DFG) for funding since parts of this research have been carried out within the Collaborative Research Center 805. The second and twelth author acknowledge support of the DFG for the EXPRESS project within the DFG priority program CoSIP (DFG-SPP 1798). The development of PolySCIP was funded by the DFG from 2012 until 2016 via the project A5 "Multicriteria Optimization" within the Collaborative Research Center 1026 "Sustainable Manufacturing".

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

We thank Miles Lubin for his support in the design and development of CSIP and SCIP.jl.

Code Contributions of the Authors

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

The technical improvements have been contributed by GG (Cycle detection 2.1.1, Independent components 2.1.3), GH (Clique components 2.1.2), GG and BM (Propagation loop improvements 2.1.4), TG, GG, BM (Enforcement of relaxation solution 2.1.5), JW (Partial solution 2.1.6, Randomization 2.1.7) and RG (Hash tables 2.1.8). The updates to the presolving in SCIP presented in Section 2.2 has been the work of GG (Presolving levels, stuffing) and DW (Complementary slackness). New primal heuristics described in Section 2.3 have been developed by GH (Graph induced neighborhood search, LP face), GG (Locks) and JW (Partial solutions). Sections 2.4 and 2.5 detailing the extension to the reoptimization plugin and the experiments with conflict analysis respectively is the work of JW. The KKT reformulation for MBQP detailed in Section 2.6.1 has been contributed by TF. The Multi-start heuristic for MINLP in Section 2.6.2 and the NLOBBT propagator presented in Section 2.6.3 are the work of BM. Section 2.6.4 details the work of FS on outer approximation cuts. The internal parallelization for SCIP that is presented in Section 2.7 is the work of RG and SJM. The development of ConcurrentSCIP presented in Sections 2.7.2- 2.7.3 was the work of RG. The new interfaces for SCIP detailed in Section 3 have been developed by BM (JSCIPOpt) and FS and RS (SCIP.jl). The updates to scaling in SoPlex that are presented in Section 4.1 is the work of MM and DR. Solution polishing detailed in Section 4.2 has been developed by MM. The DBDS presented in Section 4.3 is the work of SJM. Section 5 details the updates to UG contributed by YS. The contributions for GCG, presented in Section 6, are the work of ML, CP and JTW. The extensions to SCIP presented in Section 7 are the work of DR (SCIP-Jack), SS (PolySCIP) and TG (SCIP-SDP).

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. LP relaxation modification and cut selection in a MIP solver, June 11 2013. US Patent 8,463,729.

[4] T. Achterberg, R. E. Bixby, Z. Gu, E. Rothberg, and D. Weninger. Presolve reductions in mixed integer programming. Technical Report 16-44, ZIB, Takustr. 7, 14195 Berlin, 2016.

[5] T. Achterberg, T. Koch, and A. Martin. MIPLIB 2003. *Operations Research Letters*, 34(4):1–12, 2006.

[6] T. Achterberg, T. Koch, and A. Martin. Miplib 2003. *Operations Research Letters*, 34(4):361–372, 2006.

[7] 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, Berlin, Heidelberg, 2013.

[8] C. S. Adjiman, I. P. Androulakis, and C. A. Floudas. A global optimization method, αBB, for general twice-differentiable constrained NLPs – II. Implementation and computational results. *Computers & Chemical Engineering*, 22(9):1159–1179, 1998.

[9] C. S. Adjiman, S. Dallwig, C. A. Floudas, and A. Neumaier. A global optimization method, αBB, for general twice-differentiable constrained NLPs – I. Theoretical advances. *Computers & Chemical Engineering*, 22(9):1137–1158, 1998.

[10] M. Aigner, A. Biere, and C. Kirsch. Analysis of portfolio-style parallel sat solving on current multi-core architectures. In *Submitted*, 2013.

[11] A. Atamtürk, G. L. Nemhauser, and M. W. Savelsbergh. Conflict graphs in solving integer programming problems. *European Journal of Operational Research*, 121(1):40–55, 2000.

[12] T. Balyo, A. Biere, M. Iser, and C. Sinz. SAT race 2015. *Artif. Intell.*, 241:45–65, 2016.

[13] T. Balyo, P. Sanders, and C. Sinz. Hordesat: A massively parallel portfolio SAT solver. *CoRR*, abs/1505.03340, 2015.

[14] D. P. Baron. Quadratic programming with quadratic constraints. *Naval Research Logistics Quarterly*, 19(2):253–260, 1972.

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

[16] H. P. Benson and E. Sun. Outcome space partition of the weight set in multiobjective linear programming. *Journal of Optimization Theory and Applications*, 105(1):17–36, 2000.

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

[18] 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.

[19] T. Berthold. *Heuristic algorithms in global MINLP solvers*. PhD thesis, TU Berlin, 2014.

[20] T. Berthold, S. Heinz, and S. Vigerske. Extending a CIP framework to solve MIQCPs. In J. Lee and S. Leyffer, editors, *Mixed Integer Nonlinear Programming*, volume 154 of *The IMA Volumes in Mathematics and its Applications*, pages 427–444. Springer, 2011.

[21] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

[22] R. E. Bixby. A brief history of linear and mixed-integer programming computation. *Documenta Mathematica*, pages 107–121, 2012.

[23] R. E. Bixby, E. A. Boyd, and R. R. Indovina. Miplib: A test set of mixed integer programming problems. *Siam News*, 25(2):16, 1992.

[24] R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. P. Savelsbergh. An updated mixed integer programming library: MIPLIB 3.0. *Optima*, 58:12–15, 1998.

[25] P. Bonami, L. T. Biegler, A. R. Conn, G. Cornuéjols, I. E. Grossmann, C. D. Laird, J. Lee, A. Lodi, F. Margot, N. Sawaya, and A. Wächter. An algorithmic framework for convex mixed integer nonlinear programs. *Optimization Methods and Software*, 26(6):911–931, 2011.

[26] R. Borndörfer, H. Hoppmann, and M. Karbstein. A configuration model for the line planning problem. In D. Frigioni and S. Stiller, editors, *ATMOS 2013 – 13th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems*, volume 33, pages 68–79, 2013.

[27] R. Borndörfer, S. Schenker, M. Skutella, and T. Strunk. PolySCIP. In G.-M. Greuel, T. Koch, P. Paule, and A. Sommese, editors, *Mathematical Software – ICMS 2016, 5th International Congress, Proceedings*, volume 9725 of *LNCS*, Berlin, Germany, 2016. Springer.

[28] V. J. Bowman. On the relationship of the Tchebycheff norm and the efficient frontier of multiple-criteria objectives. In H. Thiriez and S. Zionts, editors, *Multiple Criteria Decision Making: Proceedings of a Conference Jouy-en-Josas, France May 21–23, 1975*, pages 76–86, Berlin, Heidelberg, 1976. Springer Berlin Heidelberg.

[29] S. Boyd and J. Dattorro. Alternating projections. *Lecture notes of EE 392 o, Stanford University, Autumn Quarter*, 2004, 2003.

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

[31] C. Buchheim and A. Wiegele. Semidefinite relaxations for non-convex quadratic mixed-integer programming. *Mathematical Programming*, 141(1):435–452, 2013.

[32] S. Burer and A. N. Letchford. Non-convex mixed-integer nonlinear programming: A survey. *Surveys in Operations Research and Management Science*, 17(2):97–106, 2012.

[33] S. Burer and D. Vandenbussche. Globally solving box-constrained nonconvex quadratic programs with semidefinite-based finite branch-and-bound. *Computational Optimization and Applications*, 43(2):181–195, 2009.

[34] R. Carvajal, S. Ahmed, G. Nemhauser, K. Furman, V. Goel, and Y. Shao. Using diversification, communication and parallelism to solve mixed-integer linear programs. *Oper. Res. Lett.*, 42(2):186–189, Mar. 2014.

[35] Cbc: An LP-based branch-and-cut library. https://projects.coin-or.org/Cbc.

[36] P. Celis. *Robin Hood Hashing*. PhD thesis, University of Waterloo, Waterloo, Ont., Canada, 1986.

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

[38] J. Chen and S. Burer. Globally solving nonconvex quadratic programming problems via completely positive programming. *Mathematical Programming Computation*, 4(1):33–52, 2011.

[39] B. V. Cherkassky and A. V. Goldberg. On implementing the push—relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[40] G. Cornuéjols and M. Dawande. A class of hard small 0-1 programs. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 284–293. Springer, 1998.

[41] A. R. Curtis and J. K. Reid. On the automatic scaling of matrices for Gaussian elimination. *IMA Journal of Applied Mathematics*, 10:118–124, 1972.

[42] CUTEr. A constrained and unconstrained testing environment, revisited. http://www.cuter.rl.ac.uk.

[43] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.

[44] G. B. Dantzig. *Linear programming and extensions*. Princeton Univ. Press, Princeton, NJ, 1963.

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

[46] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997.

[47] I. Dunning, J. Huchette, and M. Lubin. JuMP: A modeling language for mathematical optimization. *arXiv preprint arXiv:1508.01982*, 2015.

[48] I. Elhallaoui, A. Metrane, G. Desaulniers, and F. Soumis. An improved primal simplex algorithm for degenerate linear programs. *INFORMS Journal on Computing*, 23(4):569–577, 2011.

[49] I. Elhallaoui, A. Metrane, F. Soumis, and G. Desaulniers. Multi-phase dynamic constraint aggregation for set partitioning type problems. *Mathematical Programming*, 123:345–370, 2010.

[50] I. Elhallaoui, D. Villeneuve, F. Soumis, and G. Desaulniers. Dynamic aggregation of set-partitioning constraints in column generation. *Operations Research*, 53(4):632–645, 2005.

[51] FICO. FICO Xpress Optimization Suite. See http://www.fico.com/en/products/fico-xpress-optimization-suite.

[52] T. Fischer. *Branch-and-cut for complementarity and cardinality constrained linear programs*. PhD thesis, Technical University of Darmstadt, 2017.

[53] M. Fischetti, F. Glover, and A. Lodi. The feasibility pump. *Mathematical Programming*, 104(1):91–104, 2005.

[54] M. Fischetti and A. Lodi. Local branching. *Mathematical Programming*, 98:23–47, 2003.

[55] M. Fischetti, A. Lodi, M. Monaci, D. Salvagnin, and A. Tramontani. Improving branch-and-cut performance by random sampling. *Mathematical Programming Computation*, 8(1):113–132, 2016.

[56] M. Fischetti and M. Monaci. Proximity search for 0-1 mixed-integer convex programming. *J. Heuristics*, 20(6):709–731, 2014.

[57] J. J. Forrest and D. Goldfarb. Steepest-edge simplex algorithms for linear programming. *Mathematical Programming*, 57(1):341–374, 1992.

[58] R. Fourer. Notes on the dual simplex method. Draft report 9, Northwestern University, 1994.

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

[60] K. Fukuda and A. Prodon. Double description method revisited. In M. Deza, R. Euler, and I. Manoussakis, editors, *Combinatorics and Computer Science: 8th Franco-Japanese and 4th Franco-Chinese Conference Brest, France, July 3–5, 1995 Selected Papers*, pages 91–111, Berlin, Heidelberg, 1996. Springer.

[61] T. Gally, M. E. Pfetsch, and S. Ulbrich. A framework for solving mixed-integer semidefinite programs. Technical report, Optimization Online, 2016.

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

[63] G. Gamrath, T. Fischer, T. Gally, A. M. Gleixner, G. Hendel, T. Koch, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, S. Schenker, R. Schwarz, F. Serrano, Y. Shinano, S. Vigerske, D. Weninger, M. Winkler, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 3.2. Technical Report 15-60, ZIB, Takustr. 7, 14195 Berlin, 2016.

[64] G. Gamrath, B. Hiller, and J. Witzig. Reoptimization techniques for MIP solvers. In E. Bampis, editor, *Experimental Algorithms: 14th International Symposium, SEA 2015, Paris, France, June 29 – July 1, 2015, Proceedings*, volume 9125 of *LNCS*, pages 181–192. Springer, 2015.

[65] 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*, pages 1–66, 2016.

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

[67] G. Gamrath and M. E. Lübbecke. Experiments with a generic Dantzig-Wolfe decomposition for integer programs. In P. Festa, editor, *Experimental Algorithms*, volume 6049 of *Lecture Notes in Computer Science*, pages 239–252, Berlin, Heidelberg, 2010. Springer.

[68] B. Geißler, A. Martin, A. Morsi, and L. Schewe. The MILP-relaxation approach. In T. Koch, B. Hiller, M. E. Pfetsch, and L. Schewe, editors, *Evaluating Gas Network Capacities*, pages 103–122. MOS-SIAM Series on Optimization, 2015.

[69] S. Ghosh. *Heuristics for Integer Programs*. Canadian theses. University of Alberta (Canada), 2007.

[70] A. M. Gleixner. Factorization and update of a reduced basis matrix for the revised simplex method. Technical Report 12-36, ZIB, Takustr. 7, 14195 Berlin, 2012.

[71] A. M. Gleixner, T. Berthold, B. Müller, and S. Weltge. Three enhancements for optimization-based bound tightening. *Journal of Global Optimization*, pages 1–27, 2016.

[72] A. M. Gleixner and S. Weltge. Learning and propagating lagrangian variable bounds for mixed-integer nonlinear programming. In C. Gomes and M. Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18–22, 2013. Proceedings*, pages 355–361, Berlin, Heidelberg, 2013. Springer.

[73] GLOBALLib. GAMS World global optimization library. http://www.gamsworld.org/global/globallib/globalstat.htm.

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

[75] R. L. Gottwald. Experiments with a Parallel Portfolio of SCIP Solvers. Master's thesis, Freie Universität Berlin, 2016.

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

[77] Gurobi. Gurobi Optimization. See http://www.gurobi.com/.

[78] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a parallel SAT solver. *JSAT*, 6(4):245–262, 2009.

[79] J. Hao and J. B. Orlin. A faster algorithm for finding the minimum cut in a directed graph. *J. Algorithms*, 17(3):424–446, 1994.

[80] J. D. Hogg and J. A. Scott. On the effects of scaling on the performance of Ipopt. *CoRR*, abs/1301.7283, 2013.

[81] ILOG, Inc. ILOG CPLEX: High-performance software for mathematical programming and optimization. See http://www.ilog.com/products/cplex/.

[82] D. S. Johnson, M. Minkoff, and S. Phillips. The prize collecting steiner tree problem: Theory and practice. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms – SODA*, pages 760–769, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

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

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

[85] 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.

[86] M. K. Kozlov, S. P. Tarasov, and L. G. Khachiyan. The polynomial solvability of convex quadratic programming. *USSR Computational Mathematics and Mathematical Physics*, 20(5):223–228, 1980.

[87] J. Kronqvist, A. Lundell, and T. Westerlund. The extended supporting hyperplane algorithm for convex mixed-integer nonlinear programming. *Journal of Global Optimization*, 64(2):249–272, 2016.

[88] L. Ladányi, T. K. Ralphs, and L. E. Trotter Jr. Branch, cut, and price: Sequential and parallel. In *Computational combinatorial optimization*, pages 223–260. Springer, 2001.

[89] P. L'Ecuyer and R. Simard. Testu01: Ac library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)*, 33(4):22, 2007.

[90] LEMON: Library for efficient modeling and optimization in networks. https://lemon.cs.elte.hu/trac/lemon.

[91] T. C. Lin and D. Vandenbussche. Box-constrained quadratic programs with fixed charge variables. *Journal of Global Optimization*, 41(1):75–102, 2007.

[92] J. T. Linderoth and T. K. Ralphs. Noncommercial software for mixed-integer linear programming. *Integer programming: theory and practice*, 3:253–303, 2005.

[93] A. Lodi, T. K. Ralphs, F. Rossi, and S. Smriglio. Interdiction branching. Technical report, Technical Report OR/09/10, DEIS, Universita di Bologna, 2009.

[94] M. Lübbecke and C. Puchert. Primal heuristics for branch-and-price algorithms. In D. Klatte, H.-J. Lüthi, and K. Schmedders, editors, *Operations Research Proceedings 2011*, pages 65–70. Springer, Berlin, 2012.

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

[96] M. Lubin and I. Dunning. Computing in operations research using Julia. *INFORMS Journal on Computing*, 27(2):238–248, 2015.

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

[98] G. Marsaglia and A. Zaman. The KISS generator. Technical report, Department of Statistics, University of Florida, 1993.

[99] G. P. McCormick. Computability of global solutions to factorable nonconvex programs: Part I – Convex underestimating problems. *Mathematical Programming*, 10(1):147–175, 1976.

[100] MINLP library 2. http://gamsworld.org/minlp/minlplib2. revision number r277.

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

[102] J. J. Moré. Generalizations of the trust region problem. *Optimization Methods and Software*, 2(3-4):189–209, jan 1993.

[103] MOSEK ApS. *The MOSEK C optimizer API manual. Version 8.0 (Revision 45)*, 2016.

[104] T. S. Motzkin and E. G. Straus. Maxima for graphs and a new proof of a theorem of Turán. *Canadian Journal of Mathematics*, 17:533–540, 1965.

[105] L. M. Munguía, G. Oxberry, and D. Rajan. PIPS-SBB: A parallel distributed-memory branch-and-bound algorithm for stochastic mixed-integer programs. In *IEEE Workshop on Parallel Computing and Optimization, IPDPS, Chicago*, 2016.

[106] J. Omer, M. Towhidi, and F. Soumis. The positive edge pricing rule for the dual simplex. *Computers & Operations Research*, 61:135–142, 2015.

[107] T. Pajor, E. Uchoa, and R. F. Werneck. A robust and scalable algorithm for the Steiner problem in graphs. *CoRR*, abs/1412.2787, 2014.

[108] P. M. Pardalos and S. A. Vavasis. Quadratic programming with one negative eigenvalue is NP-hard. *Journal of Global Optimization*, 1(1):15–22, 1991.

[109] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[110] N. Ploskas and N. Samaras. The impact of scaling on simplex type algorithms. In *Proceedings of the 6th Balkan Conference in Informatics*, BCI '13, pages 17–22, New York, NY, USA, 2013. ACM.

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

[112] QPlib2014. Quadratic programming library 2014. http://www.lamsade.dauphine.fr/QPlib2014/doku.php.

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

[114] T. Ralphs and M. Galati. DIP – decomposition for integer programming. https://projects.coin-or.org/Dip.

[115] D. Rehfeldt and T. Koch. Transformations for the prize-collecting Steiner tree problem and the maximum-weight connected subgraph problem to SAP. Technical Report 16-36, ZIB, Takustr. 7, 14195 Berlin, 2016.

[116] D. Rehfeldt, T. Koch, and S. Maher. Reduction techniques for the prize-collecting steiner tree problem and the maximum-weight connected subgraph problem. Technical Report 16-47, ZIB, Takustr. 7, 14195 Berlin, 2016.

[117] S. Richter, V. Alvarez, and J. Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proc. VLDB Endow.*, 9(3):96–107, Nov. 2015.

[118] R. T. Rockafellar. *Convex analysis*. Princeton University Press, 1970.

[119] D. M. Ryan and B. A. Foster. An integer programming approach to scheduling. *Computer scheduling of public transport urban passenger vehicle and crew scheduling*, pages 269–280, 1981.

[120] H. Ryoo and N. Sahinidis. A branch-and-reduce approach to global optimization. *Journal of Global Optimization*, 8:107–138, 1996.

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

[122] S. Schenker, R. Borndörfer, and M. Skutella. A novel partitioning of the set of non-dominated points. Technical Report 16-55, ZIB, Takustr. 7, 14195 Berlin, 2016.

[123] S. Schenker, J. G. Steingrímsson, R. Borndörfer, and G. Seliger. Modelling of bicycle manufacturing via multi-criteria mixed integer programming. In *Procedia CIRP*, volume 26, pages 276–280, 2015.

[124] T. Schlechte and R. Borndörfer. Balancing efficiency and robustness – a bi-criteria optimization approach to railway track allocation. In M. Ehrgott, B. Naujoks, T. J. Stewart, and J. Wallenius, editors, *Multiple Criteria Decision Making for Sustainable Energy and Transportation Systems*, volume 634. Springer Berlin Heidelberg, 2010.

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

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

[127] 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.

[128] Y. Shinano, S. Heinz, S. Vigerske, and M. Winkler. FiberSCIP – a shared memory parallelization of SCIP. Technical Report ZR 13-55, Zuse Institute Berlin, 2013.

[129] L. Smith, J. Chinneck, and V. Aitken. Improved constraint consensus methods for seeking feasibility in nonlinear programs. *Computational Optimization and Applications*, 54(3):555–578, 2013.

[130] T. Strunk. Multikriterielle ganzzahlige Programme und ihre Lösung durch Gewichtsraumalgorithmen. Master's thesis, Technische Universität Berlin, 2014.

[131] SWIG. www.swig.org. version number 3.0.10.

[132] M. Thorup. High speed hashing for integers and strings. *CoRR*, abs/1504.06804, 2015.

[133] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.

[134] TinyCThread. http://tinycthread.github.io/.

[135] Z. Ugray, L. Lasdon, J. C. Plummer, F. Glover, J. Kelly, and R. Martí. A multistart scatter search heuristic for smooth NLP and MINLP problems. In R. Sharda, S. Voß, C. Rego, and B. Alidaee, editors, *Metaheuristic Optimization via Memory and Evolution: Tabu Search and Scatter Search*, pages 25–57, Boston, MA, 2005. Springer US.

[136] F. Vanderbeck. Branching in branch-and-price: a generic scheme. *Mathematical Programming*, 130(2):249–294, 2005.

[137] A. F. Veinott. The supporting hyperplane method for unimodal programming. *Operations Research*, 15(1):147–152, feb 1967.

[138] T. Westerlund and F. Pettersson. An extended cutting plane method for solving convex MINLP problems. *Computers & Chemical Engineering*, 21:131–136, 1995.

[139] J. Witzig, T. Berthold, and S. Heinz. Experiments with conflict analysis in mixed integer programming. Technical Report 16-63, ZIB, Takustr. 7, 14195 Berlin, 2016.

[140] R. Wong. A dual ascent approach for Steiner tree problems on a directed graph. *Mathematical Programming*, 28:271–287, 1984.

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

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

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

[144] Y. Ye. *Interior Point Algorithms: Theory and Analysis*. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, New York, 1997.

[145] A. Zanette, M. Fischetti, and E. Balas. Lexicography and degeneracy: can a pure cutting plane algorithm work? *Mathematical Programming*, 130(1):153–176, 2011.

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

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

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

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

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

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

[152] Zimpl: Zuse Institut Mathematical Programming Language. http://zimpl.zib.de/.