The SCIP Optimization Suite 8.0

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

April 8, 2022

Abstract The SCIP Optimization Suite provides a collection of software packages for mathematical optimization centered around the constraint integer programming framework SCIP. This paper discusses enhancements and extensions contained in version 8.0 of the SCIP Optimization Suite. Major updates in SCIP include improvements in symmetry handling and decomposition algorithms, new cutting planes, a new plugin type for cut selection, and a complete rework of the way nonlinear constraints are handled. Additionally, SCIP 8.0 now supports interfaces for Julia as well as Matlab. Further, UG now includes a unified framework to parallelize all solvers, a utility to analyze computational experiments has been added to GCG, dual solutions can be postsolved by PaPILO, new heuristics and presolving methods were added to SCIP-SDP, and additional problem classes and major performance improvements are available in SCIP-Jack.

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

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

*Extended author information is available at the end of the paper. The work for this article has been partly conducted within the *Research Campus MODAL* funded by the German Federal Ministry of Education and Research (BMBF grant number 05M14ZAM) and has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 773897. It has also been partly supported by the German Research Foundation (DFG) within the Collaborative Research Center 805, Project A4, and the EXPRESS project of the priority program CoSIP (DFG-SPP 1798), the German Research Foundation (DFG) within the project HPO-NAVI (project number 391087700).

1 Introduction

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

- the modeling language ZIMPL [57],
- the presolving library PAPILO for linear and mixed-integer linear programs, a new addition in version 7.0 of the SCIP Optimization Suite [**?**],
- the simplex-based linear programming solver SOPLEX [119],
- the constraint integer programming solver SCIP [3], which can be used as a fast standalone solver for mixed-integer linear and nonlinear programs and a flexible branch-cut-and-price framework,
- the automatic decomposition solver GCG [33], and
- the UG framework for parallelization of branch-and-bound solvers [101].

All six tools can be downloaded in source code and are freely available for members of noncommercial and academic institutions. They are accompanied by several extensions for solving specific problem-classes such as the award-winning Steiner tree solver SCIP-JACK [35] and the mixed-integer semidefinite programming solver SCIP-SDP [32]. This paper describes the new features and enhanced algorithmic components contained in version 8.0 of the SCIP Optimization Suite.

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

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

defined by $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, ℓ, $u \in \overline{\mathbb{R}}^n$, and the index set of integer variables $\mathcal{I} \subseteq \mathcal{N} \coloneqq \{1, \ldots, n\}$. The usage of $\overline{\mathbb{R}} \coloneqq \mathbb{R} \cup \{-\infty, \infty\}$ allows for variables that are free or bounded only in one direction (we assume that variables are not fixed to $\pm\infty$).

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

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

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

SCIP is not restricted to solving MILPs and MINLPs, but is a framework for solving *constraint integer programs* (CIPs), a generalization of the former two problem classes. The introduction of CIPs was motivated by the modeling flexibility of constraint programming and the algorithmic requirements of integrating it with efficient solution techniques available for MILPs. Later on, this framework allowed for an integration of MINLPs as well. Roughly speaking, CIPs are finite-dimensional optimization problems with arbitrary constraints and a linear objective function that satisfy the following

property: if all integer variables are fixed, the remaining subproblem must form a linear or nonlinear program.

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

The core of SCIP coordinates a central branch-cut-and-price algorithm. The methods for processing constraints of a given type are implemented in a corresponding constraint handler, and advanced methods like primal heuristics, branching rules, and cutting plane separators can be integrated as plugins with a pre-defined interface. SCIP comes with many such plugins needed to achieve a good MILP and MINLP performance. In addition to plugins supplied as part of the SCIP distribution, new plugins can be created by users. This basic design and solving process is described in more detail by Achterberg [2].

By design, SCIP interacts closely with the other components of the SCIP Optimization Suite. Optimization models formulated in Zimpl can be read by SCIP. PaPILO provides an additional fast and effective presolving procedure that is called from a SCIP presolver plugin. The linear programs (LPs) solved repeatedly during the branch-cut-and-price algorithm are by default optimized with SoPlex. Interfaces to several external LP solvers exist, and new interfaces can be added by users. GCG extends SCIP to automatically detect problem structure and generically apply decomposition algorithms based on the Dantzig-Wolfe or the Benders' decomposition scheme. And finally, the default instantiations of the UG framework use SCIP as a base solver in order to perform branch-and-bound in parallel computing environments with shared or distributed memory architectures.

New Developments and Structure of the Paper This paper focuses on two main aspects. The first one is to explain the changes and progress made in the solving process of SCIP and analyze the resulting improvements on MILP and MINLP instances, both in terms of performance and robustness. A performance comparison of SCIP 8.0 against SCIP 7.0 is carried out in Section 2. Improvements to the core of SCIP are presented in Section 3 and include

- a new framework for handling nonlinear constraints,
- symmetry handling on general variables and improved orbitope detection,
- a new separator for mixing cuts,
- improvements to decomposition-based heuristics and the Benders decomposition framework, and
- a new type of plugins for cut selection, and several technical improvements.

A more detailed explanation of the changes to the MINLP solving process and the new expression framework is given in Section 4. Improvements to the default LP solver SoPlex and presolver PaPILO are explained in Section 5 and 6 respectively. This aspect will be of interest to the optimization community working on methods and algorithms related to these building blocks and to practitioners willing to understand the performance they observe on their particular instances.

The second aspect of this paper is to present the evolving possibilities for working with the SCIP Optimization Suite 8.0 for optimization practitioners. This includes improvements and changes to the interfaces in Section 7 and the modeling language Zimpl in Section 8; to SCIP extensions specialized for other computational settings such

as distributed computing with UG in Section 9 and Dantzig-Wolfe decompositions with GCG in Section 10; and finally to SCIP extensions for particular problem classes such as the mixed-integer semidefinite solver SCIP-SDP in Section 11 and the Steiner tree solver SCIP-Jack in Section 12.

2 Overall Performance Improvements for MILP and MINLP

SCIP is used extensively to solve mixed-integer linear and nonlinear programs out of the box. In this section, we present computational experiments conducted by running SCIP without parameter tuning or algorithmic variations to assess the performance changes since the 7.0.0 release. We detail below the methodology and results of these experiments.

The indicators of interest to compare the two versions of SCIP on a given subset of instances are the number of solved instances, the shifted geometric mean of the number of branch-and-bound nodes, and the shifted geometric mean of the solving time. The *shifted geometric mean* of values $t_1, \ldots, t_n$ is

$$\Big(\prod_{i=1}^{n}(t_i + s)\Big)^{1/n} - s.$$

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

2.1 Experimental Setup

We use the SCIP Optimization Suite 7.0.0 as the baseline, including SoPlex 5.0.0 and PaPILO 1.0.0, and compare it with SCIP Optimization Suite 8.0 including SoPlex 6.0 and PaPILO 2.0. Both were compiled using GCC 7.5, use Ipopt 3.12.13 as NLP subsolver built with the MUMPS 4.10.0 numerical linear algebra solver, CppAD 20180000.0 as algorithmic differentiation library, and bliss 0.73 for graph automorphisms to detect symmetry in MIPs. The time limit was set to 7200 seconds in all cases.

The MILP instances are selected from the MIPLIB 2003, 2010, and 2017 [40] and COR@L instance sets, including all instances previously solved by SCIP 7.0.0 with at least one of five random seeds or newly solved by SCIP 8.0 with at least one of five random seeds; this amounts to 347 instances. The MINLP instances are similarly selected from the MINLPLib[1] with newly solvable instances added to the ones previously solved by SCIP 7 for a total of 113 instances.

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

2.2 MILP Performance

The results of the performance runs on MILP instances are presented in Table 1. The changes introduced with SCIP 8.0 improved the performance on MILPs both in terms of number of solved instances and shifted geometric mean of the time. Furthermore, the difference in terms of geometric mean time is starker on harder instances, with an improvement of up to 52% on instances taking more than 1000 seconds to solve. The

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

Table 1: Performance comparison for MILP instances

		SCIP 8.0.0+SoPlex 6.0.0			SCIP 7.0.0+SoPlex 5.0.0			relative	
Subset	instances	solved	time	nodes	solved	time	nodes	time	nodes
all	1708	1478	231.3	3311	1445	271.3	4107	1.17	1.24
affected	1475	1424	173.8	2843	1391	209.7	3611	1.21	1.27
[0,tilim]	1529	1478	154.4	2512	1445	184.6	3167	1.20	1.26
[1,tilim]	1470	1419	185.9	2870	1386	223.8	3647	1.20	1.27
[10,tilim]	1361	1310	248.1	3612	1277	303.1	4661	1.22	1.29
[100,tilim]	1000	949	537.1	7270	916	702.6	10262	1.31	1.41
[1000,tilim]	437	386	1566.2	17973	353	2383.1	31707	1.52	1.76
diff-timeouts	135	84	2072.7	19597	51	5062.1	69354	2.44	3.54
both-solved	1394	1394	119.9	2048	1394	133.8	2330	1.12	1.14

Table 2: Performance comparison for MINLP

		SCIP 8.0.0+SoPlex 6.0.0			SCIP 7.0.0+SoPlex 5.0.0			relative	
Subset	instances	solved	time	nodes	solved	time	nodes	time	nodes
all	558	454	39.1	2427	435	45.7	1845	1.17	0.76
affected	487	438	23.5	1748	419	28.4	1456	1.21	0.83
[0,tilim]	503	454	21.7	1585	435	25.9	1326	1.19	0.84
[1,tilim]	375	326	56.1	3994	307	71.0	3113	1.27	0.78
[10,tilim]	293	244	121.6	7450	225	169.3	5393	1.39	0.72
[100,tilim]	195	146	307.6	14204	127	433.9	6696	1.41	0.47
[1000,tilim]	153	104	466.9	23425	85	565.3	8382	1.21	0.36
diff-timeouts	117	68	451.4	29142	49	461.8	6275	1.02	0.22
both-solved	386	386	8.2	609	386	10.4	806	1.27	1.32

improvement is more limited on both-solved instances that were solved by both solvers, for which the relative improvement is only of 11%. This indicates that the overall speedup is due to newly solved instances more than to improvement on instances that were already solved by SCIP 7.0.

2.3 MINLP Performance

With the major revision of the handling of nonlinear constraints, the performance of SCIP on MINLPs has changed a lot on the instance set compared to SCIP 7.0. The MINLP performance results are summarized in Table 2. On all subsets of the instances selected by runtime, more instances are solved by SCIP 8.0 than by SCIP 7.0. Furthermore, SCIP 8.0 solves the instances for each of these subsets with a shorter shifted geometric mean time even though it produces more nodes in the branch-and-bound tree.

On the 382 instances solved by both versions, SCIP 8.0 requires fewer nodes and less time. The number of instances solved by only one of the two versions (diff-timeouts) is much higher than reported in previous release reports with similar experiments, with 66 instances newly solved by SCIP 8.0 and 46 instances previously solved that SCIP 8.0 did not succeed on.

A finer comparison of the two SCIP versions on additional subsets of instances is provided in Table 3. Instances are split according to mixed-integer and continuous, nonconvex and convex problems. They are classified as mixed-integer if at least one integer or binary constraint is present in the original problem. The convexity of instances is identical to the information provided on the MINLPLib website.

Table 3 shows that SCIP 8.0 brings most significant improvements for nonconvex problems with 41 more instances solved and a drastic speedup factor of 3.54 on the purely continuous nonconvex problems. Performance has, however, degraded on convex problems with 21 instances that are not solved anymore and the shifted geometric mean runtime more than tripled.

Table 3: Detailed MINLP performance comparison

Subset			SCIP 8+SoPlex 6		SCIP 7+SoPlex 5		relative
convexity	integrality	instances	solved	time	solved	time	time
nonconvex	continuous	95	88	66.60	71	235.86	3.54
nonconvex	mixed-integer	320	258	236.43	244	315.74	1.34
nonconvex	total	415	346	186.43	315	295.92	1.59
convex	continuous	5	5	0.14	5	0.06	0.45
convex	mixed-integer	160	118	329.37	134	167.62	0.51
convex-no-syn	mixed-integer	130	108	209.04	108	190.70	0.91
convex-syn-only	mixed-integer	30	10	1685.14	26	87.01	0.05
convex	total	165	123	310.84	139	159.76	0.51

As can be seen in the table, this is mostly due to worse performance on a specific group of instances, the `syn` group of instances. The `syn` group includes specifically instances `syn40m04h`, `rsyn0840m03h`, `rsyn0820m02m`, `syn20h`, `syn30m03h`, and `rsyn0840m04h`. The solving time for `syn` instances has degraded significantly on SCIP 8.0 while the degradation is moderate on other convex instances. The much higher time on the `syn` instances explains alone the degradation on the total convex subset. We presume that the new expression simplification at the moment obfuscates some structure on some instances of the `syn` group that was exploited with SCIP 7.0.

An MINLP performance evaluation that focuses only on the changes in handling nonlinear constraints is given in Section 4.14.

3 SCIP

3.1 A New MINLP Framework

The SCIP 8.0 release comes with a major change in the way that nonlinear constraints are handled. The main motivation for this change is twofold: First, it aims at increasing the reliability of the solver and alleviating a numerical issue that arose from problem reformulations and led to SCIP returning solutions that are feasible in the reformulated problem, but infeasible in the original problem. Second, the new design of the nonlinear framework reduces the ambiguity of expression and structure types by implementing different plugin types for low-level expression types that define expressions, and high-level structure types that add functionality for particular, often overlapping structures. Finally, a number of new features for improving the solver's performance on MINLPs were introduced. A detailed description of the changes can be found in Sections 4 and 3.2.2.

3.2 Improvements in Symmetry Handling

Symmetries are well-known to have an adverse effect on the performance of MILP and MINLP solvers, because symmetric subproblems are treated repeatedly without providing new information to the solver. For this reason, there exist different methods to handle symmetries in SCIP. Until version 7.0, SCIP was only able to handle symmetries in MILPs. With the release of SCIP 8.0 also symmetries in MINLPs can be handled. Furthermore, the release of SCIP 8.0 features several algorithmic enhancements of existing as well as the implementation of further symmetry handling methods. In the following, we describe the kind of symmetries SCIP can handle and list the techniques used in SCIP 7.0. Afterwards, we describe the novel symmetry handling methods and highlight algorithmic enhancements.

Let us start with some preliminary remarks. For a permutation γ of the variable index set $\{1, \dots, n\}$ and a vector $x \in \mathbb{R}^n$, we define $\gamma(x) = (x_{\gamma^{-1}(1)}, \dots, x_{\gamma^{-1}(n)})$. We say

that γ is a *symmetry* of (MINLP) if the following holds: $x \in \mathbb{R}^n$ is feasible for (MINLP) if and only if $\gamma(x)$ is feasible, and $c^\top x = c^\top \gamma(x)$. The set of all symmetries forms a group $\bar{\Gamma}$, the *symmetry group* of (MINLP). Since computing $\bar{\Gamma}$ is NP-hard, see Margot [73], one typically refrains from handling all symmetries. Instead, one only computes a subgroup Γ of $\bar{\Gamma}$ that keeps the constraint system of (MINLP) invariant. Computing this *formulation group* Γ for MILPs can be accomplished by computing symmetries of an auxiliary graph, see Salvagnin [94]. In SCIP 8.0, the already existing routine for computing symmetries of MILPs has been extended to handle also nonlinear constraints. To detect symmetries of the auxiliary graphs, SCIP uses the graph isomorphism package BLISS [50].

3.2.1 Previously Existing Symmetry Handling Methods in SCIP

SCIP 7.0 used two paradigms to handle symmetries of binary variables: a constraint-based approach or the pure propagation-based approach *orbital fixing* [71, 72, 82]. The constraint-based approach is implemented via three different constraint handler plugins to deal with different kinds of matrix symmetries. The *symresack* constraint handler [48] provides separation and propagation routines for general permutations γ, whereas the *orbisack* constraint handler [51] uses specialized separation and propagation methods if γ is a composition of 2-cycles. The *orbitope* constraint handler [14, 48] handles symmetries of special subgroups of Γ. These subgroups are required to act on binary matrices and to be able to reorder their columns arbitrarily. Moreover, if the variables affected by the corresponding permutations or groups interact with set packing or partitioning constraints in a certain way, all constraint handlers provide specialized separation and propagation mechanisms to find stronger cutting planes and reductions [47, 52, 53]. The common ground of these constraint handlers is that they enforce solutions that are lexicographically maximal in their orbit of symmetric solutions.

The integer parameter `misc/usesymmetry` can be used to enable/disable these two methods. In SCIP 7.0, the parameter ranged between 0 and 3, where the Bit 1 enables/disables the usage of the constraint-based approach and Bit 2 enables/disables orbital fixing. If the group Γ is a product group $\Gamma = \Gamma_1 \otimes \cdots \otimes \Gamma_k$, the variables affected by one factor of Γ are not affected by any other factor. In this case, one can apply different symmetry handling methods for the different factors. The sets of variables affected by the different factors are called the *components* of Γ. Thus, if both methods are enabled, SCIP searches for independent components of the symmetry group Γ and, depending on structural properties of the component, either uses cutting planes or orbital fixing: if a component can be completely handled by orbitopes, SCIP uses orbitopes and orbital fixing otherwise; see the SCIP Optimization Suit 7.0 release report [36] for further details.

3.2.2 Symmetry Detection Extended to Nonlinear Constraints

In SCIP 8.0, symmetry detection has been extended to handle nonlinear constraints.

The detection of permutation-based symmetries is performed by analyzing expression graphs as first proposed by Liberti [61]. The detected automorphisms are then projected onto problem variables, which yields a permutation group.

For more details, see Wegscheider [117].

3.2.3 New Symmetry Handling Methods

One drawback of the mentioned approaches is that they can only handle symmetries of binary variables, but not of general integer or continuous variables. Moreover, SCIP 7.0 can only detect orbitopes when a component of Γ can be completely handled by orbitopes,

but not if some part of the component allows applying orbitopes. In SCIP 8.0, both issues are resolved by the implementation of further symmetry handling methods and a refined detection and handling mechanism for orbitopes.

Symmetry Handling of General Variables To handle symmetries of general variables, we have implemented symmetry handling inequalities derived from the Schreier-Sims table (SST cuts) as described by Salvagnin [95]; see also Liberti and Ostrowski [62] for a more general version. These inequalities are defined using the following procedure. Let Γ be the symmetry group of (MINLP) and let $A(\Gamma) = \{i \in \{1, \dots, n\} : \exists \gamma \in \Gamma \text{ with } \gamma(i) \neq i\}$ be its set of affected variables. Select a variable index $\ell \in A(\Gamma)$ and compute its orbit $O = \{\gamma(\ell) : \gamma \in \Gamma\}$. We call ℓ the *leader* of its orbit. Afterwards, replace the initial group Γ by the stabilizer group $\{\gamma \in \Gamma : \gamma(\ell) = \ell\}$, compute the set of affected variables, and iterate the procedure of selecting a leader and replacing groups by their stabilizer until the set of affected variables becomes empty. At termination, we are given a list of leaders $\ell_1, \dots, \ell_k$ with associated orbits $O_1, \dots, O_k$. Salvagnin [95] shows that the inequalities

$$x_{\ell_i} \geq x_j, \qquad\qquad j \in O_i,\ i \in \{1, \dots, k\},$$

can be used to handle symmetries of general variables.

The above procedure allows to select the orbit leaders arbitrarily. In SCIP 8.0, several rules exist to select the leader using the parameters `propagating/symmetry/<Xyz>`, where `Xyz` is one of the following: `sstleadervartype`, `sstleaderule`, `ssttiebreakrule`, and `sstmixedcomponents`. The bitset `sstleadervartype` controls which (combinations of) variable types can be used as leaders; if several variable types are allowed, SCIP selects the one with the most affected variables. The rule `sstleaderule` determines whether the first or last variable (according to SCIP's variable ordering) in an orbit shall be used as leader. If a binary variable shall be used as leader, the parameter also allows to use the number of other binary variables it is in conflict with as a selection criterion. We say that two binary variables are in conflict if not both can be 1 simultaneously. The rule `ssttiebreakrule` selects a leader whose orbit is as small or as large as possible, or that contains the most conflicting binary variables. Finally, `sstmixedcomponents` controls whether SST cuts are allowed to be added to components of Γ that contain variables of different types. By default, we allow adding SST cuts for non-binary variables whose orbit is as small as possible. We select the first variable per orbit as leader and allow different variable types in a component.

Since SST cuts extend the class of previous symmetry handling methods, the range of parameter `misc/usesymmetry` has been extended to $\{0, \dots, 7\}$, where Bit 4 controls whether SST cuts are enabled. This in particular means that SST cuts can be used in combination with other symmetry handling methods. Below we will describe how SST cuts are used when also other symmetry handling methods are active.

Improved Orbitope Detection As mentioned previously, SCIP 7.0 uses orbitopes for a component of Γ only if all permutations within the component form an orbitope structure. This, however, can be rather restrictive as illustrated next. Consider the problem of coloring an undirected graph $G = (V, E)$ with k colors. Every feasible coloring can be encoded by a matrix $X \in \{0, 1\}^{V \times k}$, where $X_{vi} = 1$ if and only if node v is colored by color i. We can transfer the coloring X into another equivalent coloring Y by taking an arbitrary permutation π of $\{1, \dots, k\}$ and defining $Y_{vi} = X_{v\pi(i)}$. That is, the symmetry group Γ of the coloring problem can reorder the columns of binary matrices arbitrarily, and thus, allows the application of orbitopes as indicated above. If the graph G is symmetric, however, Γ will also contain permutations that reorder the rows of X according to automorphisms of G. Since these row permutations interact with the variables affected by column permutations, they form a common component. Hence,

not all permutations within this component are permutations necessary for an orbitope and the detection routine of SCIP 7.0 will not recognize the applicability of orbitopes.

In SCIP 8.0 we have refined the orbitope detection routine to be able to heuristically find such hidden orbitopes. In the following, we call such orbitopes *suborbitopes*, because they are defined by a subgroup of a component. To explain the procedure, note that a subgroup of a component defines an orbitope for matrix X if the component contains permutations that swap adjacent columns of X, see Hojny and Pfetsch [48]. Such a swap of two columns is a permutation that decomposes into 2-cycles. Therefore, our routine sieves all permutations P from a component that has such a decomposition. Then, we iteratively build a set of permutations $Q \subseteq P$ that define an orbitope or several independent orbitopes. Initially, $Q = \emptyset$ and we check, one after another, whether adding $\gamma \in P$ to Q allows to define independent orbitopes. If this is possible, Q is updated; otherwise, we continue with the next permutation in P and discard γ. To check whether we can add γ to Q, we maintain a list of the orbitopes defined by the permutations in Q so far. Then, γ is added to Q either if the variables affected by γ are not contained in any of the already known orbitopes, or γ adds a new column to an already existing orbitope, or it merges two existing orbitopes.

If a component can not completely be handled by a single orbitope, there might exist variables that are not contained in one of the detected orbitopes, or several independent suborbitopes are found that are linked via permutations not contained in Q. To partially add the missing link, and thus to handle more symmetries, SCIP selects one of the found orbitopes with variable matrix $X \in \{0,1\}^{s \times t}$ and computes one round of SST cuts with X_{11} as leader. We refer to these inequalities as *weak* inequalities, because they weakly connect the found orbitopes without exploiting any further group structure. Besides weak inequalities, we can also add *strong* inequalities $X_{11} \geq X_{12} \geq \cdots \geq X_{1t}$ for every found orbitope. These cuts are called strong because they also exploit the group structure allowing to arbitrarily reorder the columns of the orbitope. Note that the strong inequalities are implicitly added by orbitope constraints. In some situations, however, SCIP adds strong inequalities instead of orbitopes as we explain next.

The detection of suborbitopes and application of strong and weak inequalities can be controlled via the Boolean parameter `propagating/symmetry/detectsubgroups`. If the parameter value is `TRUE` (default), SCIP searches for suborbitopes using the above mechanism. A found orbitopes is called *useful* if it has at least three columns. The reason for this classification is that an orbitope with just two columns can also be handled by orbisack constraints, which can more easily be combined with other symmetry handling constraints. Moreover, the Boolean parameters `propagating/symmetry/addstrongsbcs` and `propagating/symmetry/addweaksbcs` enable/disable whether strong inequalities are used if suborbitopes are not handled and whether weak inequalities are used to handle more group structure, respectively.

SCIP*'s Symmetry Handling Strategy* As explained above, SCIP allows to handle symmetries using different strategies depending on the parameter `misc/usesymmetry`. If a mixed strategy is used, SCIP analyzes the structure of the symmetry group's components and decides which strategy is used for which component. In one case, however, these strategies can also be combined and applied to the same component: If both symresacks and SST cuts for binary variables are enabled, SCIP computes SST cuts first. The leaders of SST cuts then play a special role, because they need to attain the largest values in their orbits. To make these cuts compatible with symresacks, one thus needs to adapt the lexicographic order used by symresacks giving the leaders the highest rank.

Similarly, if suborbitopes are detected, the orbitopes can be made compatible with symresacks for the permutations not used by the orbitopes by adapting the variable order in a specific way: the variables of the first orbitope get the highest rank in the

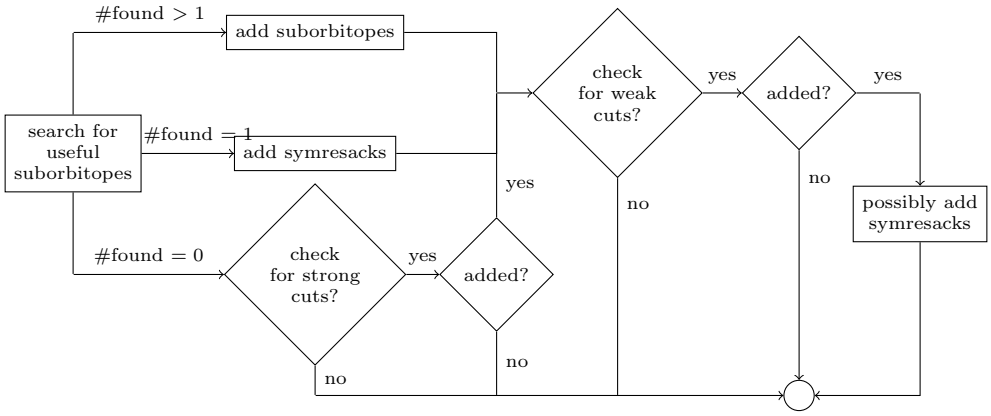

Figure 1: Rules to decide whether strong/weak inequalities are added.

lexicographic order, afterwards the variables of the succeeding orbitopes are listed, and finally the variables not contained in any orbitope are added to the lexicographic order. The exact mechanism how suborbitopes are combined with weak and strong inequalities as well as symresacks is explained in Figure 1.

SCIP's strategy to decide on which symmetry handling methods are used is carried out in the following order, depending on the enabled strategies; by default, SCIP is allowed to use all implemented symmetry handling methods (`misc/usesymmetry = 7`). First, SCIP checks whether a component can be fully handled by orbitopes or whether suborbitopes can be detected. If the component is handled by (sub)orbitopes, it gets blocked and no other symmetry handling method can be applied to this component. Second, SCIP adds SST cuts to all applicable non-blocked components and blocks these components. If the selected leaders are binary, also symresacks can applied to this component. Third, if a component has not been blocked yet, either symresacks or orbital fixing is used to handle symmetries, depending on whether orbital fixing is active.

3.2.4 Algorithmic Enhancements

Besides new symmetry handling methods, SCIP 8.0 also contains more efficient implementations of previously available methods that we describe in turn.

First, as mentioned above, orbisack constraints allow to apply stronger cutting planes or reductions if they interact with set packing or partitioning constraints in a certain way. SCIP automatically checks whether such an upgrade is possible. The implementation of this upgrade has been revised and is more efficient in SCIP 8.0.

Second, the symresack constraint handler separates so-called minimal cover inequalities for symresacks. In SCIP 7.0, we used a quadratic time separation routine for these inequalities. With the release of SCIP 8.0, these inequalities can be separated in linear time, which also improves on the almost linear running time procedure by Hojny and Pfetsch [48]. The linear time procedure makes use of the observation [48] that minimal cover inequalities for symresacks can be separated by merging connected components of an auxiliary graph. Using a disjoint-set data structure, an almost linear running time could be achieved. In our new implementation, we exploit that the graph's connected components are either paths or cycles. Merging such connected components can be realized using more efficient data structures based on a few arrays.

Finally, both the symresack and orbisack constraint handler provide routines to propagate their constraints. While the previous implementation could miss some variable fixings, the implementation in SCIP 8.0 allows to find all variable fixings that can be derived from local variable bound information.

3.2.5 Further Features

Although symresack and orbitope constraints have been available in SCIP since version 5.0, these constraints could not be parsed in any file format. With the release of SCIP 8.0, these constraints can be parsed when reading a `cip` file. Thus, users can easily tell SCIP about the symmetries that are present in their problems and how to handle them.

For a permutation γ of $\{1, \dots, n\}$ and a vector $x \in \{0,1\}^n$, a symresack constraint enforces that x is lexicographically not smaller than its permutation $\gamma(x)$. This can be encoded in a `cip` file using the line

`symresack([varName1,...,varNameN],[`$\gamma(1)$`,...,`$\gamma(n)$`]).`

Since orbisacks are symresacks for permutations that decompose into 2-cycles, this structure can directly be encoded using an $\frac{n}{2} \times 2$ matrix, where each row encodes the variables that can be interchanged. The `cip` encoding is then given by

```
fullOrbisack(varName1-1,varName1-2.varName2-1,varName2-2.  ...).
```

If users know that in each row of the orbitope matrix at most or exactly one variable can attain value 1, they can provide this information to SCIP by replacing `fullOrbisack` by `packOrbisack` or `partOrbisack`, respectively.

Finally, an orbitope constraint for a variable matrix $X \in \{0,1\}^{m \times n}$ can be encoded similarly to an orbisack by the line

```
fullOrbitope(varName1-1,...,varName1-N. ....varNameM-1,...,varNameM-N).
```

If in each row of the orbitope at most or exactly one variable can attain value 1, `fullOrbitope` can be replaced by `packOrbitope` or `partOrbitope`, respectively, to provide this information to SCIP.

3.3 Mixing Cuts

Mixing cuts [9, 44] can effectively reduce the computational time to solve MIP formulations of chance constrained programs (CCPs), especially for those in which the uncertainty appears only in the right-hand side [65, 58, 1, 121]. In order to enhance the capability of employing SCIP as a black box to solve such CCPs, SCIP 8.0 includes a new separator called `mixing`, which leverages the *variable bound relations* [2, 68] to construct mixing cuts. It is worthwhile remarking that though the development of this feature is motivated by CCPs, the mixing separator can, however, be applied for other MIPs as long as the related variable bound relations can be detected by SCIP.

Let us first review the variable bound relations in SCIP; for more details, see Achterberg [2] and the SCIP Optimization Suite 4.0 release report [68]. A variable bound relation in SCIP is a linear constraint on two variables. As such, it is of the form $y \star ax + b$ with $a, b \in \mathbb{R}$ and $\star \in \{\leq, \geq\}$. During the presolving process, SCIP derives these relations either from two-variable linear constraints or general constraints by probing [96] and stores them in a data structure called *variable bound graph*. Such relations can be used to, for example, tighten the bounds of variables through propagation [68] or enhance the MIR cuts separation [70] in the subsequent main solution process. The mixing cut separator uses a subclass of these relations, that is, those in which x is a binary variable and y is a non-binary variable. Thereby, three families of cuts are constructed which is discussed in detail in the following. For simplicity, we only consider the case that y is a continuous variable but the result can also be applied to the case that y is an integer variable.

$\geq$-Mixing Cuts Consider the variable lower bounds of variable $y \in [\ell, u]$:

$$y \geq a_i x_i + b_i,\ x_i \in \{0,1\},\ i \in \mathcal{N}. \tag{3}$$

Without loss of generality, we impose the following assumption:

$$0 < a_i \leq u - \ell \text{ and } b_i = \ell \text{ for all } i \in \mathcal{N}. \tag{A}$$

Indeed, assumption (A) can be guaranteed by applying the following preprocessing steps in order:

(i) If $a_i < 0$, variable x_i can be complemented by $1 - x_i$. If $a_i = 0$, $y \geq a_i x_i + b_i$ can be removed from (3) and $\ell' := \max\{\ell, b_i\}$ is the new lower bound for y.

(ii) If $a_i + b_i \leq \ell$, by $a_i > 0$ (from (i)), constraint $y \geq a_i x_i + b_i$ is implied by $y \geq \ell$ and hence can be removed from (3).

(iii) If $b_i > \ell$, by $a_i > 0$ (from (i)), $\ell' := b_i$ is the new lower bound for y; if $b_i < \ell$, by $a_i + b_i > \ell$ (from (ii)), relation $y \geq a_i x_i + b_i$ can be changed into $y \geq (a_i + b_i - \ell)x_i + \ell$.

(iv) If $a_i > u - \ell$, by $b_i = \ell$ (from (iii)), $x_i = 0$ must hold and constraint $y \geq a_i x_i + \ell$ can be removed from (3).

By assumption (A), (3) can be presented in normalized form:

$$y \geq a_i x_i + \ell,\ x_i \in \{0,1\},\ i \in \mathcal{N}. \tag{4}$$

Let $\{i_1, \ldots, i_s\} \subseteq \mathcal{N}$ with $s \in \mathbb{N}$ such that $a_{i_1} \leq \cdots \leq a_{i_s}$, and define $a_{i_0} := 0$. Then the $\geq$-mixing inequality [9, 44] is given by

$$y - \ell \geq \sum_{\tau=1}^{s} (a_{i_\tau} - a_{i_{\tau-1}})\, x_{i_\tau}. \tag{5}$$

$\leq$-Mixing Cuts Using a similar analysis as that in variable lower bounds, the variable upper bounds of variable y can be presented in normalized form:

$$y \leq u - a_j x_j,\ x_j \in \{0,1\},\ j \in \mathcal{M}, \tag{6}$$

where $0 < a_j \leq u - \ell$, $j \in \mathcal{M}$. Let $\{j_1, \ldots, j_t\} \subseteq \mathcal{M}$ ($t \in \mathbb{N}$) such that $a_{j_1} \leq \cdots \leq a_{j_t}$, and define $a_{j_0} := 0$. Then the $\leq$-mixing inequality [9, 44] is given by

$$y \leq u - \sum_{\tau=1}^{t} (a_{j_\tau} - a_{j_{\tau-1}})\, x_{j_\tau}. \tag{7}$$

Conflict Cuts Besides the $\geq$- and $\leq$-mixing cuts, the mixing separator also constructs conflict cuts, which are derived by jointly considering (4) and (6). To be more specific, let $i' \in \mathcal{N}$ and $j' \in \mathcal{M}$ such that $a_{i'} + \ell > u - a_{j'}$. By $y \geq a_{i'} x_{i'} + \ell$ and $y \leq u - a_{j'} x_{j'}$, variables $x_{i'}$ and $x_{j'}$ cannot simultaneously take values at one, and hence the conflict inequality

$$x_{i'} + x_{j'} \leq 1 \tag{8}$$

can be derived.

Separation Given a fractional point (x^*, y^*), the separation problem of (5), (7) or (8) asks to find an inequality violated by (x^*, y^*) or prove that no such one exists. To separate the $\geq$-mixing inequalities (5), Günlük and Pochet [44] provided the following algorithm, which selects the subset $\mathcal{S} = \{i_1, \ldots, i_\tau\} \subseteq \mathcal{N}$ such that $\sum_{\tau=1}^{s} (a_{i_\tau} - a_{i_{\tau-1}})\, x^*_{i_\tau}$ is maximized.

1. Reorder variables x_i, $i \in \mathcal{N}$, such that $x_1^* \geq x_2^* \geq \cdots \geq x_{|\mathcal{N}|}^*$.
2. Add 1 to set $\mathcal{S}$.
3. For each $i \in \mathcal{N} \backslash \{1\}$, set $\mathcal{S} := \mathcal{S} \cup \{i\}$ if $a_i > a_k$, where k is last index added into $\mathcal{S}$.
4. If the $\geq$-mixing inequality corresponding to $\mathcal{S}$ is violated by (x^*, y^*), output it.

Obviously, the above algorithm can be implemented to run in $\mathcal{O}(|\mathcal{N}| \log(|\mathcal{N}|))$. Similarly, the $\leq$-mixing inequalities (7) can also be separated in $\mathcal{O}(|\mathcal{M}| \log(|\mathcal{M}|))$. Finally, for the conflict inequalities (8), since number of them is bounded by $\mathcal{O}(|\mathcal{M}||\mathcal{N}|)$, by enumeration, they can be separated in $\mathcal{O}(|\mathcal{M}||\mathcal{N}|)$.

The performance impact of the mixing separator is neutral on the internal MIP benchmark testset. However, when applied to the chance constrained lot sizing instances used by Zhao et al. [121] (90 in total), a speedup of 20% can be observed and 15 more instances can be solved.

3.4 Primal Decomposition Heuristics

SCIP 8.0 comes with an improvement of the heuristic Penalty Alternating Direction Method (`PADM`) and introduces the new heuristic Dynamic Partition Search (`DPS`). Both heuristics explicitly require a decomposition provided by the user and therefore belong to the class of so-called *decomposition heuristics*. A decomposition consisting of $k \geq 0$ blocks is a partition

$$\mathcal{D} \coloneqq (D^{\text{row}}, D^{\text{col}}) \text{ with } D^{\text{row}} \coloneqq (D_1^{\text{row}}, \ldots, D_k^{\text{row}}, L^{\text{row}}),\ D^{\text{col}} \coloneqq (D_1^{\text{col}}, \ldots, D_k^{\text{col}}, L^{\text{col}})$$

of the rows/columns of the constraint matrix A into $k+1$ pieces each. The distinguished rows L^{row} and columns L^{col} are called *linking rows* and *linking columns*, respectively. If A is permuted according to decomposition $\mathcal{D}$, a *(bordered) block diagonal form* [18] is obtained. A detailed description of decompositions and their handling in SCIP can be found in the release report for version 7.0 [36].

3.4.1 Improvement of Penalty Alternating Direction Method

Since version 7.0, SCIP includes the decomposition heuristic Penalty Alternating Direction Method (`PADM`). For the current version `PADM` has been extended by the option to improve a found solution by reintroducing the original objective function.

This heuristic splits a MINLP as listed in (2) into several subproblems according to a given decomposition $\mathcal{D}$ with linking variables only, whereby the linking variables get copied and the differences are penalized. Then the subproblems are solved by an alternating procedure. A detailed description of penalty alternating direction methods and their practical application can be found in Geißler et al. [38] and Schewe et al. [97].

To converge faster to a feasible solution, the original objective function of each subproblem has been completely replaced by a penalty term. Since this can lead to arbitrarily bad solutions, the heuristic was extended in the following way: Initially, the original version of `PADM` runs and tries to find a feasible solution. If a feasible solution was found the linking variables are fixed to the values of this solution and each independent subproblem is solved again but now with the original objective function. In order to accelerate the solving of the reoptimization step, the already found solution is used as a warm start solution and very small solving limits are imposed. The additional reoptimization step must not take more time than was already used by the heuristic in the first step and the node limit is set to one. By setting the parameter `heuristics/padm/reoptimize` the feature of using a second reoptimization step in `PADM` can be turned on/off (default: on).

The new feature was tested on the MIPLIB 2017 [40] benchmark instances, for which decompositions are provided on the web page. If `PADM` could get called, preliminary results show that `PADM` finds a feasible solution in 15 of 31 cases. The new reoptimization step successfully improves the solution of `PADM` in 33% of these cases by 42% on average.

3.4.2 Dynamic Partition Search

With SCIP 8.0 the new decomposition heuristic Dynamic Partition Search (`DPS`) was added. It is a primal construction heuristic which requires a decomposition with linking constraints only.

The `DPS` heuristic splits a MILP as listed in (1) into several subproblems according to a decomposition $\mathcal{D}$. Thereby the linking constraints and their right/left-hand sides are also split by introducing new parameters $p_q \in \mathbb{R}^{|L^{\text{row}}|}$ for each block $q \in \{1, \dots, k\}$ and requiring that

$$\sum_{q=1}^{k} p_q = b_{[L^{\text{row}}]} \tag{9}$$

holds. To obtain information about the infeasibility of one subproblem and to speed up the solving process, slack variables $z_q \in \mathbb{R}_+^{|L^{\text{row}}|}$ are added and the objective function is replaced by a weighted sum of these slack variables. In detail, for penalty parameter $\lambda \in \mathbb{R}_{>0}^{|L^{\text{row}}|}$ each subproblem q has the form

$$\begin{aligned}
&\min \lambda^\top z_q, \\
&\text{s.t. } A_{[D_q^{\text{row}}, D_q^{\text{col}}]} x_{[D_q^{\text{col}}]} \geq b_{[D_q^{\text{row}}]}, \\
&\quad \ell_i \leq x_i \leq u_i && \text{for all } i \in \mathcal{N} \cap D_q^{\text{col}}, \\
&\quad x_i \in \mathbb{Z} && \text{for all } i \in \mathcal{I} \cap D_q^{\text{col}}, \\
&\quad A_{[L^{\text{row}}, D_q^{\text{col}}]} x_{[D_q^{\text{col}}]} + z_q \geq p_q, \\
&\quad z_q \in \mathbb{R}_+^{|L^{\text{row}}|}.
\end{aligned} \tag{10}$$

From (10), it is immediately apparent that the correct choice of p_q plays the central role. Because if p_q is chosen for each subproblem q such that the slack variables z_q take the value zero, one immediately obtains a feasible solution. For this reason, we refer to $(p_q)_{q \in \{1,\dots,k\}}$ as a *partition* of $b_{[L^{\text{row}}]}$. The goal of `DPS` is to find a feasible partition as fast as possible.

To get started, an initial partition $(p_q)_{q \in \{1,\dots,k\}}$ is chosen, which fulfills (9). Then it is checked whether this partition will lead to a feasible solution by solving k independent subproblems (10) with fixed p_q. If all subproblems have an optimal objective value of zero, a feasible solution was found and is given by the concatenation of the k subsolutions. Conversely, a lower bound on the objective function value of one subproblem of greater than zero immediately provides evidence that the current partition does not lead to a feasible solution.

If the current partition does not correspond to a feasible solution, then partition $(p_q)_{q \in \{1,\dots,k\}}$ and penalty parameter λ have to be updated: For each single linking constraint $j \in L^{\text{row}}$ the value vector z_j is subtracted from the current partition p_j and the same amount is added to all blocks with $z_{jq} = 0$, so that (9) still holds. If at least one slack variable is positive, the corresponding penalty parameter is increased. Then, the subproblems are solved again and the steps are repeated until a feasible solution is found or until a maximum number of iterations (controlled by parameter `heuristics/dps/maxiterations`) is reached.

To push the slack variables to zero and to speed up the algorithm, the original objective function has been completely replaced by a penalty term. Analogously to `PADM`

(see Section 3.4.1) it is possible to improve the found solution by reoptimizing with the original objective function. In DPS the partition instead of the linking variables is fixed. By setting parameter heuristics/dps/reoptimize this feature can be turned on/off (default: off).

The new decomposition heuristic was tested on the MIPLIB 2017 [40] benchmark instances, for which decompositions are provided on the web page. If DPS could get called, preliminary results show that DPS finds a feasible solution in 17 of 80 cases. A general performance improvement can not be shown. The main reason for these slightly disappointing results is probably that DPS requires a well-decomposable problem structure. The evaluated instances are general MILPs which do not necessarily have such a structure. However, on two instances (proteindesign121hz512p9 and 30n20b8) DPS is successful and reduces the time until the first found primal solution highly, since no other heuristic is able to construct a feasible solution at or before the root node. It is noticeable that in both instances the linking constraints contain only bounded integer variables. The heuristic probably benefits from this, since the number of usable partitions is thus countable and finite.

3.5 Benders' Decomposition

The work on the Benders' decomposition framework has moved into a research phase. As such, only minor updates and bug fixes have been completed for the framework since the release of SCIP 7.0. The most important update for the Benders' decomposition framework is the option to apply the mixed integer rounding (MIR) procedure, as described by Achterberg [2], when generating optimality cuts. The aim of applying the MIR procedure to the generated optimality cut is to potentially compute a stronger inequality.

Strengthening the classical Benders' optimality cut using the MIR procedure involves the following steps:

- Generate a classical optimality cut from the solution of the Benders' decomposition subproblem.
- Attempt to compute a flow cover cut for the generated optimality cut. This is achieved by calling SCIPcalcFlowCover. If this process is successful, replace the optimality cut with the computed flow cover cut.
- Attempt to perform the MIR procedure on the optimality cut (this could have been updated in the previous step). The MIR procedure is performed by calling SCIPcalcMIR. If the MIR procedure is successful, the optimality cut is replaced with the resulting inequality.
- Finally, SCIPcutsTightenCoefficients is executed in an attempt to tighten the coefficients of the optimality cut.

The MIR procedure is active by default. A new parameter

benders/<bendersname>/benderscut/optimality/mir,

where <bendersname> is the name of the Benders' decomposition plugin, has been added to enable/disable the MIR procedure for strengthening the Benders' optimality cuts.

3.6 Cut Selectors

The new cut selector plugin is introduced in SCIP 8.0. Users now have the ability to create their own cut selection rules and include them into SCIP. For a current summary on the state of cut selection in the literature, see Dey and Molinaro [23], and for an overview

of cutting plane measures and the improvements provided by intelligent selection, see Wesselmann and Suhl [118]. The existing rule used since SCIP 6.0 [39] has been moved to `cutselection/hybrid`. The ability to include cut selectors has also been implemented through PySCIPOpt.

3.7 Technical Improvements

Thread Safety In previous versions, SCIP contained the argument `PARASCIP` for the Make and CMake build system to make it thread-safe. This has been replaced by `THREADSAFE`, which is now true by default (`PARASCIP` still exists for backward compatibility).

Most parts of SCIP are in fact always thread-safe, but interfaces to external programs are sometimes not. For instance, for the LP-solver Gurobi, the thread-safe mode opens a new LP-environment for each thread. Other interfaces to external software may use parallelization that has to be controlled in order not to mix data from different threads, e.g., CppAD and FilterSQP. The change to thread-safe mode should not significantly affect performance.

Revision of External Memory Estimation SCIP usually uses its own internal memory functions. This allows to keep track of the used memory. If it approaches the memory limit, SCIP can switch to a memory saving mode, which, for instance, uses depth-first-search. However, memory used by external software, in particular, NLP and LP-solvers cannot easily be determined in a portable way. Therefore, the estimation of used memory in SCIP has been improved for version 8 with data-fitting as follows. The memory consumption by LP-solvers was measured using a stand-alone version on a testset of LP-relaxations. Then a linear regression with the number of constraints, variables, and nonzeros as features was computed. This current estimation uses the weights 8.5×10^{-4}, 7.6×10^{-4}, 3.5×10^{-5}, respectively, and works quite well (for SoPlex we got $R^2 = 0.99$). If NLPs are solved, the estimation is doubled.

Option to Forbid Variable Aggregation Similar to multi-aggregation, one can now forbid aggregation of a variable by calling the function `SCIPdoNotAggrVar()`. This is sometimes useful, for example, if certain constraint handlers cannot handle aggregated variables. Note, however, that this can slow down the solving process since the relaxations tend to be larger.

Debugging of Variable Uses SCIP counts the number of uses of a variable and frees a variable when its uses count reaches zero. It is therefore important to capture a variable to prevent it from being freed too early and to release a variable when it is no longer used. To assist on finding a missing or excessive capture or release of a variable, code has been added to `var.c` to print the function call stack when a variable of a specified name, optionally in a SCIP problem of a specified name, is captured or released. The code requires GCC Gnulib (`execinfo.h` in particular) and will not work on every platform.

To activate this feature, define `DEBUGUSES_VARNAME` and `DEBUGUSES_PROBNAME` in `var.c`. If the tool `addr2line` is available on the system, the printed call stacks provide more information, but its use causes a significant slowdown. Defining `DEBUGUSES_NOADDR2LINE` disables the call of this tool.

Improving Numerical Properties of Linear Inequalities When a constraint handler or separator computes a cutting plane, often its numerical properties need to be checked and possibly improved before it is added to a relaxation. Further, changing coefficients or sides of a `SCIP_ROW` may round numbers that are very close to integral values, which may invalidate a previously valid cut. To assist on carefully improving the numerical

properties of an inequality, the `SCIP_ROWPREP` datastructure has been made available, see `pub_misc_rowprep.h`. Routines are available to relax or scale linear inequalities to improve the range of coefficients and to avoid almost-integral numbers, see the paragraph "Cut cleanup" in Section 4.2.10 and Step 4 in Section 4.2.11 for more details. Note, that ranged linear constraints (both left-hand-side and right-hand-side being finite) cannot be handled.

Reader for AMPL `.nl` Files The reader for `.nl` files has been rewritten and is now included with SCIP's default plugins. See Section 7.1 for more details.

4 SCIP's New MINLP Framework

4.1 New Expressions Framework

Algebraic expressions are well-formed combinations of constants, variables, and various algebraic operations such as addition, multiplication, exponentiation, that are used to describe mathematical functions. They are often represented by a directed acyclic graph with nodes representing variables, constants, and operations and arcs indicating the flow of computation, see Figure 2 for an example.

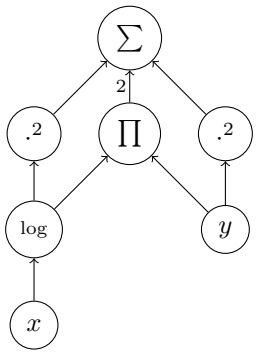

Figure 2: Expression graph for algebraic expression $\log(x)^2 + 2\log(x)y + y^2$.

With SCIP 7.0 and before, the following node types were supported in algebraic expressions:

- constant, parameter,
- variable (specified by integer index),
- addition, subtraction, multiplication, division (with two arguments),
- square, square-root, power (rational exponent), power (integer exponent), signed power ($x \mapsto \text{sign}(x)|x|^p$),
- exponentiation, natural logarithm,
- minimum, maximum, absolute value,
- sum, product, affine-linear, quadratic, signomial (with arbitrarily many arguments),
- user-defined.

The operand type "user-defined", which was introduced with SCIP 3.2 [34], brought some of the extensibility typical for SCIP plugins. However, only the most essential callbacks (evaluation, differentiation, linear under/overestimation) were defined for user-defined expressions. Thus, other routines that worked on expressions, such as simplification,

had built-in treatment for operands integrated in SCIP, but defaulted to some simple conservative behavior when a user-defined operand had to be dealt with.

Another problem with this design of expressions was the ambiguity and additional complexity due to the presence of high-level operators such as affine-linear, quadratic, and others. For example, code that did some operation on a sum had to implement the same routine for any operand that represents some form of summation (plus, sum, affine-linear, quadratic, signomial), each time dealing with a slightly different data structure.

With SCIP 8, the expression system has been completely rewritten. Proper SCIP plugins, referred to as *expression handlers*, are now used to define all semantics of an operand. These expression handlers support more callbacks than what was available for the user-defined operator before. Furthermore, much ambiguity and complexity is avoided by adding expression handlers for basic operations only. High-level structures such as quadratic functions can still be recognized, but are no longer made explicit by a change in the expression type. An expression (`SCIP_EXPR`) comprises various data, in particular the arguments of the expression, further on denoted as *children*. It can also hold the data and additional callbacks of an *expression owner*, if any. A prominent example of an expression owner is the constraint handler for nonlinear constraints, see Section 4.2, which stores data associated with the enforcement of nonlinear constraints in expressions that are used to specify nonlinear constraints. Further, due to their many use cases, a representation of the expression as a quadratic function can be stored, see Section 4.3.1 for details.

Various methods are available in the SCIP core to manage expressions (create, modify, copy, free, parse, print), to evaluate and compute derivative information at a point, to evaluate over intervals, to simplify, to identify common subexpressions, to check curvature and integrality, and to iterate over it. Many of these methods access callbacks that can be implemented by expression handlers. Some additional callbacks are used by the constraint handler for nonlinear constraints (Section 4.2). The expression handler callbacks are:

- `COPYHDLR`: include expression handler in another SCIP instance;
- `FREEHDLR`: free expression handler data;
- `COPYDATA`: copy expression data, for example, the coefficients of a linear sum;
- `FREEDATA`: free expression data;
- `PRINT`: print expression;
- `PARSE`: parse expression from string;
- `CURVATURE`: detect convexity or concavity;
- `MONOTONICITY`: detect monotonicity;
- `INTEGRALITY`: detect integrality (is value of operation integral if arguments have integral value?);
- `HASH`: hash expression using hash values of arguments;
- `COMPARE`: compare two expressions of same type;
- `EVAL`: evaluate expression (implementation of this callback is mandatory);
- `BWDIFF`: evaluate partial derivative of expression with respect to specified argument (backward derivative evaluation);
- `FWDIFF`: evaluate directional derivative of expression (forward derivative evaluation);
- `BWFWDIFF`: evaluate directional derivative of partial derivative with respect to specified argument (backward over forward derivative);
- `INTEVAL`: evaluate expression over interval;

– `ESTIMATE`: compute linear under- or overestimator of expression with respect to given bounds on arguments and a reference point;
– `INITESTIMATES`: compute one or several linear under- or overestimators of expression with respect to given bounds on arguments;
– `SIMPLIFY`: simplify expression by applying algebraic transformations;
– `REVERSEPROP`: compute bounds on arguments of expression from given bounds on expression.

The SCIP documentation provides more details on these callbacks.

Finally, for the following operators, expression handlers are included in SCIP 8.0:

– `val`: scalar constant;
– `var`: a SCIP variable (`SCIP_VAR`);
– `varidx`: a variable represented by an index; this handler is only used for interfaces to NLP solvers (NLPI);
– `sum`: an affine-linear function, $y \mapsto a_0 + \sum_{j=1}^{k} a_j y_j$ for $y \in \mathbb{R}^k$ with constant coefficients $a \in \mathbb{R}^{k+1}$;
– `prod`: a product, $y \mapsto c \prod_{j=1}^{k} y_j$ for $y \in \mathbb{R}^k$ with constant factor $c \in \mathbb{R}$;
– `pow`: a power with a constant exponent, $y \mapsto y^p$ for $y \in \mathbb{R}$ and exponent $p \in \mathbb{R}$ (if $p \notin \mathbb{Z}$, then $y \geq 0$ is required);
– `signpower`: a signed power, $y \mapsto \operatorname{sign}(y)|y|^p$ for $y \in \mathbb{R}$ and constant exponent $p \in \mathbb{R}$, $p > 1$;
– `exp`: exponentiation, $y \mapsto \exp(y)$ for $y \in \mathbb{R}$;
– `log`: natural logarithm, $y \mapsto \log(y)$ for $y \in \mathbb{R}_{>0}$;
– `entropy`: entropy, $y \mapsto \begin{cases} -y\log(y), & \text{if } y > 0, \\ 0, & \text{if } y = 0, \end{cases}$ for $y \in \mathbb{R}_{\geq 0}$;
– `sin`: sine, $y \mapsto \sin(y)$ for $y \in \mathbb{R}$;
– `cos`: cosine, $y \mapsto \cos(y)$ for $y \in \mathbb{R}$;
– `abs`: absolute value, $y \mapsto |y|$ for $y \in \mathbb{R}$.

When comparing with the list for SCIP 7.0 above, one observes that support for parameters (these behaved like constants but could not be simplified away and were modifiable) and operators "min" and "max" has been removed. Further, support for sine, cosine, and the entropy function has been added.

4.2 New Handler for Nonlinear Constraints

For SCIP 8, the constraint handler for general nonlinear constraints (`cons_nonlinear`) has been rewritten and the specialized constraint handlers for quadratic, second-order cone, absolute power, and bivariate constraints have been removed. Some of the unique functionalities of the removed constraint handlers has been reimplemented in other plugin types.

4.2.1 Motivation

An initial motivation for the rewrite of `cons_nonlinear` has been a numerical issue which is caused by the explicit reformulation of constraints in SCIP 7.0 and earlier versions.

For an example, consider the problem

$$\begin{aligned} &\min z, \\ &\text{s.t. } \exp(\ln(1000) + 1 + x\,y) \leq z, \\ &\quad x^2 + y^2 \leq 2, \end{aligned} \tag{11}$$

with optimal solution $x = -1$, $y = 1$, $z = 1000$. Previously, solving this problem with SCIP could end with the following solution report:

```
SCIP Status        : problem is solved [optimal solution found]
Solving Time (sec) : 0.08
Solving Nodes      : 5
Primal Bound       : +9.99999656552062e+02 (3 solutions)
Dual Bound         : +9.99999656552062e+02
Gap                : 0.00 %
  [nonlinear] <e1>: exp((7.9077552789821368 + (<x>*<y>)))-<z>[C] <= 0;
violation: right-hand side is violated by 0.000673453314561812
best solution is not feasible in original problem

x                                    -1.00057454873626  (obj:0)
y                                    0.999425451364613  (obj:0)
z                                     999.999656552061  (obj:1)
```

The reason that SCIP initially determined this solution to be feasible is that, in presolve, the problem gets rewritten as

$$\begin{aligned} &\min z, \\ &\text{s.t. } \exp(w) \leq z, \\ &\quad \ln(1000) + 1 + x\,y = w, \\ &\quad x^2 + y^2 \leq 2. \end{aligned} \tag{12}$$

The constraints in this transformed problem are violated by $0.4659 \cdot 10^{-6}$, $0.6731 \cdot 10^{-6}$, and $0.6602 \cdot 10^{-6}$, thus are feasible with respect to `numerics/feastol`$= 10^{-6}$, and therefore the solution is accepted by SCIP. On the MINLPLib library, the problem that a final solution is feasible for the presolved problem but violates nonlinear constraints in the original problem occurred for 7% of all instances.

Problem (11) gets rewritten as (12) for the purpose of constructing a linear relaxation. In this process, nonlinear functions are approximated by linear under- and overestimators. As the formulas that were used to compute these estimators are only available for "simple" functions (for example, convex functions, concave functions, bilinear terms), new variables and constraints were introduced to split more complex expressions into adequate form [107, 115].

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

Thus, the new constraint handler aims to preserve the original constraints by applying only transformations (simplifications) that, in most situations, do not relax the feasible space when taking tolerances into account. The reformulations that were necessary for the construction of a linear relaxation are not applied explicitly anymore, but handled implicitly by annotating the expressions that define the nonlinear constraints (here, the mysterious "data of an expression owner", see Section 4.1, comes into play). Another

advantage of this approach is a clear distinction between the variables that were present in the original problem and the variables added for the reformulation. With this information, branching is avoided on variables of the latter type. Finally, it is now possible to exploit overlapping structures in an expression simultaneously.

4.2.2 Extended Formulations

To explain the functionality of the new `cons_nonlinear`, consider MINLPs of the form

$$\begin{aligned} \min\ & c^\top x, \\ \text{s.t. } & \underline{g} \le g(x) \le \overline{g}, \\ & \underline{b} \le Ax \le \overline{b}, \\ & \underline{x} \le x \le \overline{x}, \\ & x_{\mathcal{I}} \in \mathbb{Z}^{\mathcal{I}}, \end{aligned} \tag{MINLP}$$

with $c \in \mathbb{R}^n$, $g : \mathbb{R}^n \to \overline{\mathbb{R}}^m$, $\underline{g}, \overline{g} \in \overline{\mathbb{R}}^m$, $A \in \mathbb{R}^{\tilde{m} \times n}$, $\underline{b}, \overline{b} \in \overline{\mathbb{R}}^{\tilde{m}}$, $\underline{x}, \overline{x} \in \overline{\mathbb{R}}^n$, $\mathcal{I} \subseteq \{1, \dots, n\}$, $\overline{\mathbb{R}} := \mathbb{R} \cup \{\pm\infty\}$. Further, assume that $g_i(\cdot)$ is nonlinear and specified by an expression (see Section 4.1), $i = 1, \dots, m$, $\underline{g} \le \overline{g}$, $\underline{g}_i \in \mathbb{R}$ or $\overline{g}_i \in \mathbb{R}$ for all $i = 1, \dots, m$, $\underline{b} \le \overline{b}$, $\underline{b}_i \in \mathbb{R}$ or $\overline{b}_i \in \mathbb{R}$ for all $i = 1, \dots, \tilde{m}$, and $\underline{x} \le \overline{x}$. All nonlinear constraints $\underline{g} \le g(x) \le \overline{g}$ are handled by `cons_nonlinear`, while the linear constraints are handled by `cons_linear` or its specializations. (Of course, in general, any kind of constraint that SCIP supports is allowed, but for this section only linear and nonlinear constraints are considered.) In comparison to SCIP 7.0, the specialized nonlinear constraint handlers and the distinction into a linear and a nonlinear part of a nonlinear constraint have been removed. As a consequence, all algorithms for nonlinear constraints (checking feasibility, domain propagation, separation, etc) work on expressions now.

SCIP solves problems like (MINLP) to global optimality via a spatial branch-and-bound algorithm that mixes branch-and-infer and branch-and-cut [13]. Important parts of the solution algorithm are presolving, domain propagation (that is, tightening of variable bounds), linear relaxation, and branching. For the domain propagation and linear relaxation aspects, two extended formulations of (MINLP) that are obtained by introducing *slack variables* and replacing sub-trees of the expressions that define nonlinear constraints by *auxiliary variables* are considered.

For domain propagation, the following extended formulation is considered:

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

Initially, slack variables $w_1^{\mathrm{dp}}, \dots, w_m^{\mathrm{dp}}$ are introduced and $h_i^{\mathrm{dp}} := g_i$ for $i = 1, \dots, m$. Next, for each function $h_i^{\mathrm{dp}}(x)$, subexpressions $f(x)$ may be replaced by new auxiliary variables $w_{i'}$, $i' > m$, and new constraints $h_{i'}^{\mathrm{dp}}(x) = w_{i'}^{\mathrm{dp}}$ with $h_{i'}^{\mathrm{dp}} := f$ are added. For the latter, subexpressions may be replaced again. Since auxiliary variables that replace subexpression of $h_i^{\mathrm{dp}}(x)$ always receive an index larger than $\max(m, i)$, the result is referred to by $h_i^{\mathrm{dp}}(x, w_{i+1}^{\mathrm{dp}}, \dots, w_{m^{\mathrm{dp}}}^{\mathrm{dp}})$ for any $i = 1, \dots, m^{\mathrm{dp}}$. That is, to simplify notation, w_{i+1}^{dp} is used instead of $w_{\max(i,m)+1}^{\mathrm{dp}}$. If a subexpressions that is replaced by an auxiliary variable appears in several places, then only one auxiliary variable and one

constraint is added to the extended formulation. Reindexing may be necessary to have h_i^{dp} depend on x and $w_{i+1}^{\mathrm{dp}}, \dots$ only.

The details of how subexpressions are chosen to be replaced by auxiliary variables will be discussed in Section 4.2.5. For the moment it is sufficient to assume that algorithms are available to compute interval enclosures of

$$\{h_i^{\mathrm{dp}}(x, w_{i+1}^{\mathrm{dp}}, \dots, w_{m^{\mathrm{dp}}}^{\mathrm{dp}}) : \underline{x} \le x \le \overline{x},\ \underline{w}^{\mathrm{dp}} \le w^{\mathrm{dp}} \le \overline{w}^{\mathrm{dp}}\}, \tag{13}$$

$$\{x_j : h_i^{\mathrm{dp}}(x, w_{i+1}^{\mathrm{dp}}, \dots, w_{m^{\mathrm{dp}}}^{\mathrm{dp}}) = w_i^{\mathrm{dp}} : \underline{x} \le x \le \overline{x},\ \underline{w}^{\mathrm{dp}} \le w^{\mathrm{dp}} \le \overline{w}^{\mathrm{dp}}\},\ j = 1, \dots, n, \tag{14}$$

$$\{w_j^{\mathrm{dp}} : h_i^{\mathrm{dp}}(x, w_{i+1}^{\mathrm{dp}}, \dots, w_{m^{\mathrm{dp}}}^{\mathrm{dp}}) = w_i^{\mathrm{dp}} : \underline{x} \le x \le \overline{x}, \underline{w}^{\mathrm{dp}} \le w^{\mathrm{dp}} \le \overline{w}^{\mathrm{dp}}\}, \quad j = i+1, \dots, m^{\mathrm{dp}}, \tag{15}$$

for $i = 1, \dots, m^{\mathrm{dp}}$. The variable bounds $\underline{w}^{\mathrm{dp}}, \overline{w}^{\mathrm{dp}} \in \overline{\mathbb{R}}^{m^{\mathrm{dp}}}$ are initially set to $\underline{w}_i^{\mathrm{dp}} = \underline{g}_i$, $\overline{w}_i^{\mathrm{dp}} = \overline{g}_i$, $i = 1, \dots, m$, and $\underline{w}_i^{\mathrm{dp}} = -\infty$, $\overline{w}_i^{\mathrm{dp}} = \infty$, $i = m+1, \dots, m^{\mathrm{dp}}$.

It is worth noting here that the variables w^{dp} are not actually added as SCIP variables, although this has been suggested, but merely serve notational purposes. In the context of domain propagation, only the bounds $\underline{w}^{\mathrm{dp}}$ and $\overline{w}^{\mathrm{dp}}$ are relevant and stored in the expression.

For the construction of a linear relaxation, a similar extended formulation is considered:

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

Functions $h_i^{\mathrm{lp}}(\cdot)$ are again obtained from the expressions that define functions $g_i(\cdot)$ by recursively replacing subexpressions by auxiliary variables $w_{i+1}^{\mathrm{lp}}, \dots, w_{m^{\mathrm{lp}}}^{\mathrm{lp}}$. However, it is important to note that different subexpressions may be replaced when setting up $h^{\mathrm{lp}}(\cdot)$ compared to setting up $h^{\mathrm{dp}}(\cdot)$. In fact, in contrast to (MINLP$_{\mathrm{ext}}^{\mathrm{dp}}$), it is assumed that algorithms are available to compute a linear outer-approximation of the sets

$$\{(x, w^{\mathrm{lp}}) : h_i^{\mathrm{lp}}(x, w_{i+1}^{\mathrm{lp}}, \dots, w_{m^{\mathrm{lp}}}^{\mathrm{lp}}) \lesseqgtr_i w_i^{\mathrm{lp}}, x \in [\underline{x}, \overline{x}], w^{\mathrm{lp}} \in [\underline{w}^{\mathrm{lp}}, \overline{w}^{\mathrm{lp}}]\},\ i = 1, \dots, m^{\mathrm{lp}}. \tag{16}$$

Thus, the auxiliary variables w_i^{lp}, $i = m+1, \dots, m^{\mathrm{lp}}$, can be different from w_i^{dp}, $i = m+1, \dots, m^{\mathrm{dp}}$. However, the slack variables w_i^{lp}, $i = 1, \dots, m$, can be considered as identical to w_i^{dp}. Similarly to (MINLP$_{\mathrm{ext}}^{\mathrm{dp}}$), the variable bounds $\underline{w}^{\mathrm{lp}}$, $\overline{w}^{\mathrm{lp}} \in \overline{\mathbb{R}}^{m^{\mathrm{lp}}}$ are initially set to $\underline{w}_i^{\mathrm{lp}} = \underline{g}_i$, $\overline{w}_i^{\mathrm{lp}} = \overline{g}_i$, $i = 1, \dots, m$, and $\underline{w}_i^{\mathrm{lp}} = -\infty$, $\overline{w}_i^{\mathrm{lp}} = \infty$, $i = m+1, \dots, m^{\mathrm{lp}}$. Regarding the (in)equality sense $\lesseqgtr_i$, a valid simplification would be to assume equality everywhere. For performance reasons, though, it can be beneficial to relax certain equalities to inequalities if that does not change the feasible space of (MINLP$_{\mathrm{ext}}^{\mathrm{lp}}$) when projected onto x. Therefore,

$$\lesseqgtr_i \coloneqq \begin{cases} =, & \text{if } \underline{g}_i > -\infty,\ \overline{g}_i < \infty, \\ \le, & \text{if } \underline{g}_i = -\infty,\ \overline{g}_i < \infty, \\ \ge, & \text{if } \underline{g}_i > -\infty,\ \overline{g}_i = \infty, \end{cases} \qquad \text{for } i = 1, \dots, m.$$

For $i > m$, monotonicity of expressions needs to be taken into account. This is discussed in Section 4.2.3.

In difference to ($\text{MINLP}^{\text{dp}}_{\text{ext}}$), the variables w^{lp} are added to SCIP as variables when the LP is initialized. They are marked as *relaxation-only* [36], that is, are not copied when the SCIP problem is copied and are fixed or deleted when restarting (new auxiliary variables are added for the next SCIP round).

To decide for which constraints in ($\text{MINLP}^{\text{lp}}_{\text{ext}}$) it can make sense to try to improve their linear relaxation, the value of a subexpression needs to be compared with the value for $h^{\text{lp}}_i(\cdot)$. Thus, define $\hat{h}^{\text{lp}}_i(x)$ to be the value of the subexpression that $h^{\text{lp}}_i(\cdot)$ represents if evaluated at x. Formally, for $i = 1, \ldots, m^{\text{lp}}$,

$$\hat{h}^{\text{lp}}_i(x) \coloneqq h^{\text{lp}}_i(x, w^{\text{lp}}_{i+1}, \ldots, w^{\text{lp}}_{m^{\text{lp}}}) \text{ where } w^{\text{lp}}_j \coloneqq h^{\text{lp}}_j(x, w^{\text{lp}}_{j+1}, \ldots, w^{\text{lp}}_{m^{\text{lp}}}), j = i+1, \ldots, m^{\text{lp}}.$$

Hence, $\hat{h}_i \equiv g_i$ for $i = 1, \ldots, m$.

Example Recall Figure 2 and the constraint

$$\log(x)^2 + 2\log(x)\, y + y^2 \leq 4.$$

Using structural detection algorithms (discussed in Section 4.2.5 below), SCIP may replace $\log(x)$ by an auxiliary variable w_2, since that results in a quadratic form $w_2^2 + 2w_2 y + y^2$, which is both bivariate and convex, the former being well suited for domain propagation and the latter being beneficial for linearization. Therefore, the following extended formulation ($\text{MINLP}^{\text{dp}}_{\text{ext}}$) may be constructed:

$$\begin{aligned} h^{\text{dp}}_1(x, y, w^{\text{dp}}_2) \coloneqq (w^{\text{dp}}_2)^2 + 2w^{\text{dp}}_2 y + y^2 &= w^{\text{dp}}_1, \\ h^{\text{dp}}_2(x, y) \coloneqq \log(x) &= w^{\text{dp}}_2, \\ w^{\text{dp}}_1 &\leq 4. \end{aligned}$$

($\text{MINLP}^{\text{lp}}_{\text{ext}}$) could be very similar,

$$\begin{aligned} h^{\text{lp}}_1(x, y, w^{\text{lp}}_2) \coloneqq (w^{\text{lp}}_2)^2 + 2w^{\text{lp}}_2 y + y^2 &\leq w^{\text{lp}}_1, \\ h^{\text{lp}}_2(x, y) \coloneqq \log(x) &= w^{\text{lp}}_2, \\ w^{\text{lp}}_1 &\leq 4, \end{aligned}$$

where equality has been chosen for $h^{\text{lp}}_2(x, y) = w^{\text{lp}}_2$ because $(w^{\text{lp}}_2)^2 + 2w^{\text{lp}}_2 y + y^2$ is neither monotonically increasing nor monotonically decreasing in w^{lp}_2. If, however, $y \geq 0$ and $x \geq 1$, then one may relax to $\log(x) \leq w^{\text{lp}}_2$.

Next, consider the following slight modification:

$$\log(x)^2 + 4\log(x)\, y + y^2 \leq 4.$$

SCIP may again replace $\log(x)$ by an auxiliary variable w_2, since that results in a bivariate quadratic form, but the expression is not convex anymore. SCIP may therefore decide to introduce additional auxiliary variables to disaggregate the quadratic form for the purpose of constructing a linear relaxation. Therefore, while ($\text{MINLP}^{\text{dp}}_{\text{ext}}$) would be the same as above (with coefficient 2 changed to 4), ($\text{MINLP}^{\text{lp}}_{\text{ext}}$) would be the result of associating an auxiliary variable with every node of the expression graph:

$$\begin{aligned} w^{\text{lp}}_2 + 4w^{\text{lp}}_3 + w^{\text{lp}}_4 &\leq w^{\text{lp}}_1, \\ (w^{\text{lp}}_5)^2 &\leq w^{\text{lp}}_2, \\ w^{\text{lp}}_5\, y &\leq w^{\text{lp}}_3, \\ y^2 &\leq w^{\text{lp}}_4, \\ \log(x) &= w^{\text{lp}}_5, \\ w^{\text{lp}}_1 &\leq 4. \end{aligned}$$

4.2.3 Variable and Expression Locks

For constraints that are checked for feasibility, SCIP asks the constraint handler to add down- and uplocks to the variables in the constraint. A downlock (uplock) indicates whether decreasing (increasing) the variable could render the constraint infeasible. While it would be valid to add both down- and uplocks for each variable, more precise information can be useful, for example, for the effectiveness of primal heuristic or dual presolving routines.

For constraints as in (MINLP), the monotonicity of $g(x)$ and Ax with respect to a specific variable and the finiteness of left- and right-hand sides ($\underline{g}$, $\overline{g}$, $\underline{b}$, $\overline{b}$) decides which locks should be added. While for Ax it is sufficient to check the sign of matrix entries, the monotonicity of $g(x)$ can sometimes be deduced by analyzing the expression that defines $g(x)$. Since monotonicity of $g(x)$ may depend on variable values, variable bounds should be taken into account when deriving monotonicity information and variable locks.

To derive locks for variables, the down- and uplocks for variables are generalized to expressions. That is, in each expression e a number of down- and uplocks (although they are referred to as negative and positive locks in the code) are stored, which indicate the number of constraints that could become infeasible when the value of e is decreased or increased. For variable-expressions, these down- and uplocks are then exactly the required down- and uplocks of the corresponding variables.

To start, take a constraint $\underline{g}_j \leq g_j(x) \leq \overline{g}_j$ and assume that the expression that defines $g_j(x)$ is given as $\tilde{g}(f_1(x), f_2(x), \ldots)$ for some operand $\tilde{g}$ and (sub)expressions $f_1, f_2, \ldots$. If $\overline{g}_j < \infty$, then increasing the value of $\tilde{g}$ could render the constraint infeasible, so an uplock is added to $\tilde{g}$. Analogously, if $\underline{g}_j > -\infty$, then decreasing the value of $\tilde{g}$ could render the constraint infeasible, so a downlock is added to $\tilde{g}$.

Next, these locks are "propagated" to the children $f_1, f_2, \ldots$. First, the monotonicity of $\tilde{g}$ with respect to a child f_k is checked by use of the `MONOTONICITY` callback of the expression handler for $\tilde{g}$. If $\tilde{g}$ is monotonically increasing in f_k, then increasing f_k could render those constraints infeasible that could become infeasible if $\tilde{g}$ is increased and decreasing f_k could render those constraints infeasible that could become infeasible when $\tilde{g}$ is decreased. Therefore, down- and uplocks stored for $\tilde{g}$ are added to the down- and uplocks, respectively, of f_k. If $\tilde{g}$ is monotonically decreasing in f_k, then increasing f_k would decrease $\tilde{g}$ and decreasing f_k would increase $\tilde{g}$. Therefore, the downlocks of $\tilde{g}$ are added to the uplocks of f_k and the uplocks of $\tilde{g}$ are added to the downlocks of f_k. Finally, if no monotonicity of $\tilde{g}$ in f_k could be concluded, then the sum of down- and uplocks of $\tilde{g}$ are added to both the down- and uplocks of f_k.

This procedure is applied for all expressions $f_1, f_2, \ldots$ and recursively to their successors. When a variable expression is encountered, then the down- and uplocks in the variable expression are added to the down- and uplocks of the variable. Therefore, in difference to linear and many other types of constraints in SCIP, a variable in a single constraint can get several down- or uplocks if it appears several times.

When constraints need to be "unlocked", the same procedure is run, but down- and uplocks are subtracted instead of added. To avoid that, due to tightened variable bounds, different monotonicity information is used when removing locks, the calculated monotonicity information is stored (removed) in an expression when it is locked the first time (unlocked the last time).

For an example, consider again the expression from Figure 2 and the constraint $\log(x)^2 + 2\log(x)\, y + y^2 \leq 4$. Assume further that $\underline{x} = 0$, $\overline{x} = 1$, and $\overline{y} = 0$. The locks for x and y are deduced as follows, see also Figure 3:

1. One uplock and no downlock are assigned to the sum-node, because the constraint has a finite right-hand side and no left-hand side.
2. Since every coefficient in the sum is nonnegative, every child of the sum is assigned one uplock and no downlock.

3. $\log(x)^2$ is monotonically decreasing in $\log(x)$ because $\log(x) \leq 0$, so that the uplock of $\log(x)^2$ is added to the downlocks of $\log(x)$.
4. $2\log(x)\, y$ is monotonically decreasing in $\log(x)$ because $2y \leq 0$, so the uplock of $2\log(x)\, y$ is added to the downlocks of $\log(x)$.
5. $2\log(x)\, y$ is monotonically decreasing in y because $2\log(x) \leq 0$, so the uplock of $2\log(x)\, y$ is added to the downlocks of y.
6. y^2 is monotonically decreasing in y because $y \leq 0$, so the uplock of y^2 is added to the downlocks of y.
7. $\log(x)$ is monotonically increasing in x, so the downlocks of $\log(x)$ are added to the downlocks of x.

Thus, eventually both x and y receive 2 downlocks, one for each appearance of the variables in the expression. Presolvers, primal heuristics, or other plugins of SCIP may now use the information that increasing the value of these variables in any feasible solution does not render this constraint infeasible.

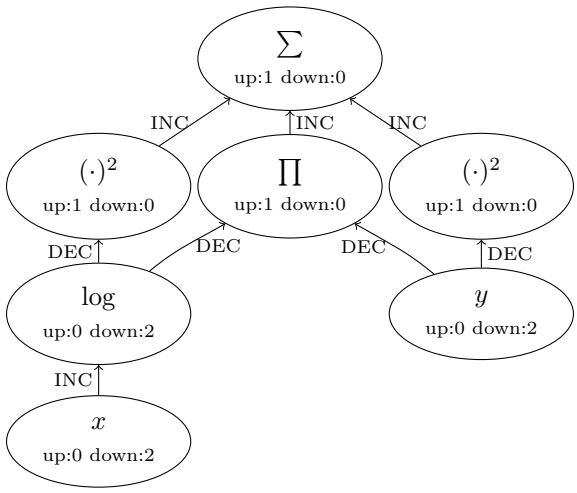

Figure 3: Propagation of locks through expression graph. INC/DEC specifies the monotonicity of parent w.r.t. child.

4.2.4 Nonlinear Handler

The construction of the extended formulations requires algorithms that analyze an expression for specific structures, for instance, quadratic or convex subexpressions as in the previous example. Following the spirit of the plugin-oriented design of SCIP, these algorithms are not hardcoded into `cons_nonlinear`, but are added as separate plugins, referred to as *nonlinear handlers*. Next to detecting structures in expressions, nonlinear handlers can also provide domain propagation and linear relaxation algorithms that act on these structures. These plugins have to interact tightly with `cons_nonlinear` and nonlinear constraints. Therefore, in difference to other plugins in SCIP, nonlinear handlers are managed by `cons_nonlinear` and not the SCIP core.

In fact, `cons_nonlinear` acts both as a handler for nonlinear constraints and as a "core" for the management and enforcement of the extended formulations ($\text{MINLP}^{\text{dp}}_{\text{ext}}$) and ($\text{MINLP}^{\text{lp}}_{\text{ext}}$). As a constraint handler, it checks nonlinear constraints for feasibility, adds them to the NLP relaxation, applies various presolving operations (see Section 4.2.7), handles variable locks, and more. When it comes to domain propagation, separation,

and enforcement of nonlinear constraints (see Sections 4.2.8–4.2.11), the constraint handler decides for which constraints in the extended formulations domain propagation or separation should be tried and calls corresponding routines in nonlinear handlers. When separation fails in enforcement, the constraint handler also selects a branching variable from a list of candidates that has been assembled by nonlinear handlers.

Since domain propagation, separation, and enforcement is partially "outsourced" into nonlinear handlers, a certain similarity of nonlinear handler callbacks to constraint handler callbacks is not surprising. A nonlinear handler can provide the following callbacks:

- `COPYHDLR`: include nonlinear handler in another SCIP instance;
- `FREEHDLRDATA`: free nonlinear handler data;
- `FREEEXPRDATA`: free expression-specific data of nonlinear handler;
- `INIT`: initialization;
- `EXIT`: deinitialization;
- `DETECT`: analyze a given expression ($h_i^{\mathrm{dp}}(\cdot)$ and/or $h_i^{\mathrm{lp}}(\cdot)$) for a specific structure and decide whether to contribute in domain propagation for $h_i^{\mathrm{dp}}(\cdot) = w_i^{\mathrm{dp}}$ or linear relaxation of $h_i^{\mathrm{lp}}(\cdot) \lesseqgtr_i w_i^{\mathrm{lp}}$ (implementation of this callback is mandatory);
- `EVALAUX`: evaluate expression with respect to auxiliary variables in descendants, that is, compute $h_i^{\mathrm{lp}}(x, w_{i+1}^{\mathrm{lp}}, \ldots, w_{m^{\mathrm{lp}}}^{\mathrm{lp}})$;
- `INTEVAL`: evaluate expression with respect to current bounds on variables, that is, compute an interval enclosure of (13);
- `REVERSEPROP`: tighten bounds on descendants, that is, compute interval enclosures of (14) and (15) and update bounds $\underline{x}_j$, $\overline{x}_j$, $\underline{w}_j$, $\overline{w}_j$ accordingly;
- `INITSEPA`: initialize separation data and add initial linearization of (16) to the LP relaxation;
- `EXITSEPA`: deinitialize separation data;
- `ENFO`: given a point $(\hat{x}, \hat{w})$, create a bound change or add a cutting plane that separates this point from the feasible set; usually, this routine tries to improve the linear relaxation of $h_i^{\mathrm{lp}}(x, w_{i+1}^{\mathrm{lp}}, \ldots, w_{m^{\mathrm{lp}}}^{\mathrm{lp}}) \lesseqgtr_i w_i^{\mathrm{lp}}$; if neither a bound change nor a cutting plane was found, register variables for which reducing their domain might help to make separation succeed;
- `ESTIMATE`: given a point $(\hat{x}, \hat{w})$, compute a linear under- or overestimator of function $h_i^{\mathrm{lp}}(x, w_{i+1}^{\mathrm{lp}}, \ldots, w_{m^{\mathrm{lp}}}^{\mathrm{lp}})$ that is as tight as possible in $(\hat{x}, \hat{w})$ and valid with respect to either the local or global bounds on x and w^{lp}; further, register variables for which reducing their domain might help to produce a tighter estimator.

More details on the exact input and output of these callbacks is given in the SCIP documentation.

4.2.5 Constructing Extended Formulations

The extended formulations ($\mathrm{MINLP}_{\mathrm{ext}}^{\mathrm{dp}}$) and ($\mathrm{MINLP}_{\mathrm{ext}}^{\mathrm{lp}}$) are constructed simultaneously by processing one nonlinear constraint of (MINLP) at a time (however, common subexpressions that are shared among different constraints are processed only once). For a constraint $\underline{g}_i \leq g(x) \leq \overline{g}_i$, $i \in \{1, \ldots, m\}$, for which domain propagation is enabled (which it is by default), a slack variable w_i^{dp} and a constraint $h_i^{\mathrm{dp}}(x) = w_i^{\mathrm{dp}}$ with $h_i^{\mathrm{dp}} \equiv g_i$ are added to ($\mathrm{MINLP}_{\mathrm{ext}}^{\mathrm{dp}}$). If, additionally, separation or enforcement is enabled (which they are by default) and SCIP is not in presolve, then the slack variable w_i^{lp} and the

constraint $h_i^{\text{lp}}(x) \lesseqgtr_i w_i^{\text{lp}}$ with $h_i^{\text{lp}} \equiv g_i$ are added[2] to (MINLP$_{\text{ext}}^{\text{lp}}$). Thereby, $\lesseqgtr_i$ is decided according to the finiteness of $\underline{g}_i$ and $\overline{g}_i$ as shown above.

Next, the `DETECT` callback of each nonlinear handler is called for the expression that defines h_i^{dp} and h_i^{lp}. The callback is informed if domain propagation and/or separation is required or if it was already provided by some other nonlinear handler. The nonlinear handler then analyzes the expression and returns whether it wants to participate in (i) domain propagation for $h_i^{\text{dp}}(x) = w_i^{\text{dp}}$, (ii) separation for $h_i^{\text{lp}}(x) \leq w_i^{\text{lp}}$, and/or (iii) separation for $h_i^{\text{lp}}(x) \geq w_i^{\text{lp}}$. Furthermore, the nonlinear handler can also introduce auxiliary variables for subexpressions, that is, transform $h_i^{\text{dp}}(x)$ into $h_i^{\text{dp}}(x, w_{i+1}^{\text{dp}}, \ldots)$, $h_i^{\text{lp}}(x)$ into $h_i^{\text{lp}}(x, w_{i+1}^{\text{lp}}, \ldots)$, and add constraints like $h_{i+1}^{\text{dp}}(x) = w_{i+1}^{\text{dp}}$ and $h_{i+1}^{\text{lp}}(x) \lesseqgtr_{i+1} w_{i+1}^{\text{lp}}$. The sense $\lesseqgtr_{i+1}$ is decided according to the expression locks (see Section 4.2.3) that have been deduced for h_{i+1}^{dp}:

$$\lesseqgtr_{i+1} := \begin{cases} =, & \text{if there are both down- and uplocks for } h_{i+1}^{\text{dp}}, \\ \leq, & \text{if there are no downlocks for } h_{i+1}^{\text{dp}}, \\ \geq, & \text{if there are no uplocks for } h_{i+1}^{\text{dp}}. \end{cases}$$

Note that this definition is consistent with the definition of $\lesseqgtr_j$, $j = 1, \ldots, m$, in Section 4.2.2 as downlocks (uplocks) are added to g_j if and only if $\underline{g}_j > -\infty$ ($\overline{g}_j < \infty$), see Section 4.2.3.

After processing h_i^{dp} and h_i^{lp}, subexpressions thereof are processed in a depth-first manner. Whenever a subexpression is visited that is associated with a constraint $h_{i'}^{\text{dp}}(x) = w_{i'}^{\text{dp}}$ or $h_{i'}^{\text{lp}}(x) \lesseqgtr_{i'} w_{i'}^{\text{lp}}$, the `DETECT` callback of each nonlinear handler is called again, this time for the expression that defines $h_{i'}^{\text{dp}}$ or $h_{i'}^{\text{lp}}$. This way, extended formulations are built for each constraint $\underline{g}_i \leq g_i(x) \leq \overline{g}_i$ in a recursive manner by utilizing the structure detection algorithms of all available nonlinear handlers.

For an example, recall again the expression from Figure 2 and assume that the initial extended formulations are $\log(x)^2 + 2\log(x)\,y + y^2 = w_1^{\text{dp}}$ and $\log(x)^2 + 2\log(x)\,y + y^2 \geq w_1^{\text{lp}}$. A nonlinear handler that inspects the sum-expression may decide that it can provide domain propagation for the expression if the log-expression was replaced by a variable. Thus, an auxiliary variable w_2^{dp} as well as a constraint $w_2^{\text{dp}} = \log(x)$ are introduced and $h_1^{\text{dp}}(x) = \log(x)^2 + 2\log(x)\,y + y^2$ is changed to $h_1^{\text{dp}}(x, w_2^{\text{dp}}) = (w_2^{\text{dp}})^2 + 2w_2^{\text{dp}} y + y^2$. Similarly, the same or another nonlinear handler may decide that it can provide a linear relaxation for the inequality if $\log(x)$ was replaced by a variable. It will introduce an auxiliary variable w_2^{lp} and a constraint $w_2^{\text{lp}} = \log(x)$. Then it will change $\log(x)^2 + 2\log(x)\,y + y^2 \geq w_1^{\text{lp}}$ to $(w_2^{\text{lp}})^2 + 2w_2^{\text{lp}}\,y + y^2 \geq w_1^{\text{lp}}$. In addition, the nonlinear handler then indicates that tight bounds for w_2^{lp} and y are required to compute the linear relaxation. This again initiates the introduction of an auxiliary variable w_2^{dp} and a constraint $w_2^{\text{dp}} = \log(x)$, given that they were not existing already.

4.2.6 Nonlinear Handler "Default"

To ensure that there always exist routines that can provide domain propagation and linear relaxation for an expression, the "fallback" nonlinear handler `default` is available. This nonlinear handler resorts to callbacks of expression handlers (see Section 4.1) to provide the necessary functionalities. However, while nonlinear handlers are usually meant to handle larger parts of an expression, the methods implemented by the expression handlers

[2] To be exact, extended formulations are not created explicitly and slack variables are not created at this state, but the top of the expression g_i in the nonlinear constraints is marked for propagation and/or separation. Variables w_i^{lp} are added in SCIP when the LP relaxation is initialized. For simplicity, these technicalities are omitted here.

are limited to the immediate children of an expression and thus have a rather myopic view on the expression. Therefore, the `DETECT` callback of the default nonlinear handler is called with a low priority. It then decides whether it contributes domain propagation or linear relaxation depending on what other nonlinear handler have declared before. If the nonlinear handler decides to contribute, it will introduce auxiliary variables w^{dp} and/or w^{lp} for all immediate children of the current expression. The other callbacks of the default nonlinear handler, in particular `EVALAUX`, `INTEVAL`, `REVERSEPROP`, `ESTIMATE`, can then utilize the corresponding "myopic" callbacks of the expression handlers.

4.2.7 Presolve

Simplify The simplify callbacks of expression handlers are called to bring the expressions into a canonical form. For example, recursive sums and products are flattened and fixed or aggregated variables are replaced by constants or sums of active variables. See the documentation of function `SCIPsimplifyExpr()` for a more exhaustive list of applied simplifications.

Forbid Multiaggregation For variables that appear nonlinearly, multiaggregation is forbidden. This aims to prevent that a simple term like x^3 will be expanded into a possible long polynomial when x is multiaggregated. The linear constraint handler currently does not account for such effects when deciding whether a variable should be multiaggregated.

Common Subexpressions Subexpressions that appear several times are identified and replaced by a single expression. This also ensures that every variable is represented by only one variable-expression across all constraints and that for expressions that appear in several nonlinear constraints at most one auxiliary variable is introduced in the extended formulations. However, sums that are part of other sums are currently not identified, since in the canonical form no sum can have a sum as a child. The same holds for products. The `HASH` and `COMPARE` callback of the expression handlers are used to identify common subexpressions.

Scaling For constraints for which the expression is a sum (which it always is if there is a constant factor different from 1.0), it is ensured that the number of terms with positive coefficients is at least the number of terms with negative coefficients by scaling the constraint with -1. If there are as many positive as negative coefficients, then it is ensured that the right-hand side is not $+\infty$. This canonicalization step can be useful for the next point.

Merge Constraints Nonlinear constraints that share the same expression are merged.

Constraint Upgrading Upgrades to other constraint types are checked. Most importantly, nonlinear constraints that are linear after simplification are replaced by constraints that are handled by `cons_linear`. Further, constraints that can be written as $(x - a_x) \cdot (y - a_y) = 0$ with x and y binary variables and $a_x, a_y \in \{0, 1\}$ are replaced by setpacking constraints.

Linearization of Binary Products Products of binary variables are linearized. This is done in a way that is similar to previous SCIP versions [115], but the consideration of cliques is new:

- In the simplest case, a product $\prod_i x_i$ is replaced by a new variable z and a constraint of type "and" is added that models $z = \bigwedge_i x_i$. The "and"-constraint handler will then separate a linearization of this product [16].

– Optionally, for a product of only two binary variables, xy, the linearization can be added directly as linear constraints ($x \geq z$, $y \geq z$, $x + y \leq 1 + z$).
– For a product of two binary variables, xy, it is checked whether x (or its negation) and y (or its negation) are contained in a common clique. Taking this information into account allows for simpler linearizations of xy. For example, x and y being in a common clique implies $x + y \leq 1$ and thus $xy = 0$. Analogously, $x + (1 - y) \leq 1$ gives $xy = x$, $(1 - x) + y \leq 1$ gives $xy = y$, and $(1 - x) + (1 - y) \leq 1$ gives $xy = x + y - 1$.
– Replacing every product in a large quadratic term $\sum_{i,j} Q_{ij} x_i x_j$ by a new variable and constraint can increase the problem size enormously. SCIP therefore checks whether there exist sums of the form $x_i \sum_j Q_{ij} x_j$ ($Q_{ij} \neq 0$) with at least 50 terms and replaces them by a single variable z_i and the linearization

$$\begin{aligned} \underline{Q} x_i &\leq z_i, \\ z_i &\leq \overline{Q} x_i, \\ \underline{Q} &\leq \sum_j Q_{ij} x_j - z_i + \underline{Q} x_i, \\ \overline{Q} &\geq \sum_j Q_{ij} x_j - z_i + \overline{Q} x_i, \end{aligned}$$

where $\underline{Q} \coloneqq \sum_j \min(0, Q_{ij})$, $\overline{Q} \coloneqq \sum_j \max(0, Q_{ij})$. This usually gives a looser LP relaxation as when each product $x_i x_j$ would be replaced individually, but has the advantage that less variables and constraints need to be introduced. Variable z_i is marked to be implicit integer if all coefficients Q_{ij} are integer. Variables x_i that appear in the highest number of bilinear terms are prioritized.

Identification of Integrality For constraints that can be written as $\sum_i a_i f_i(x) + by = c$, $b \neq 0$, it is checked whether the variable type of y can be changed to *implicitly integer*. Storing the information that a continuous variable can take only integer values in a feasible solution can be useful in the solving process, for example, when branching on y. To change the type of y, the following conditions need to be satisfied: y is of continuous type, $\frac{a_i}{b} \in \mathbb{Z}$, $\frac{c}{b} \in \mathbb{Z}$, and $f_i(x) \in \mathbb{Z}$ for solutions that satisfy integrality requirements of (MINLP) ($x_{\mathcal{I}} \in \mathbb{Z}^{\mathcal{I}}$). To determine the latter, the `INTEGRALITY` callback of expression handlers is used.

Implicit Discreteness [45] It is checked whether some variables can be restricted to be at one of their bounds. At the moment, the method looks for a non-binary variable x with finite bounds, without coefficient in the objective function, and that appears in only one constraint. Furthermore, this constraint needs to be a polynomial inequality where x appears only in monomials of the form $c_k x^{2k}$ with $k \in \mathbb{N}$ or other monomials where x has exponent 1. If all coefficients c_k have the same sign, then the constraint function is convex ($c_k > 0$) or concave ($c_k < 0$) in x. Finally, if in addition, the constraint has an infinite right-hand side (when $c_k > 0$) or an infinite left-hand side (when $c_k < 0$), then x can be restricted to be in $\{\underline{x}, \overline{x}\}$. This is valid because any feasible solution with $x \in (\underline{x}, \overline{x})$ can be transformed into another feasible solution with same objective function value by moving x to one of its bounds.

If $\underline{x} = 0$ and $\overline{x} = 1$, then x is transformed into a binary variable. Otherwise, a bound disjunction constraint $(x \leq \underline{x}) \vee (x \geq \overline{x})$ is added. This "upgrade" of continuous variables to discrete ones has been shown to be particularly effective for box-QP instances.

Identification of Unlocked Linear Variables Since SCIP supports linear objective functions only, problems with a nonlinear objective function are reformulated by the readers of and interfaces to SCIP into one with a linear objective function ($\min f(x)$ becomes

$\min z$ s.t. $f(x) \leq z$), To ensure feasibility of such artificial constraints, nonlinear constraints are checked for a variable x_i, $i \in \{1, \ldots, n\}$, that appears linearly and which value could be increased or decreased in a solution without the risk of violating other constraints (see also Section 4.2.3). When a solution candidate violates a nonlinear constraint where such a variable x_i has been identified, the constraint handler postprocesses this solution by adjusting the value of x_i such that the constraint becomes feasible. This modified solution is then passed on to primal heuristic "trysol", which will suggest it to the SCIP core the next time this primal heuristic is run.

Bound Tightening Domain propagation is run (see Section 4.2.8) to tighten variable bounds and identify redundant or always-infeasible constraints ($g([\underline{x}, \overline{x}]) \subseteq [\underline{g}, \overline{g}]$ or $g([\underline{x}, \overline{x}]) \cap [\underline{g}, \overline{g}] = \emptyset$). The extended formulation ($\text{MINLP}^{\text{dp}}_{\text{ext}}$) for domain propagation is constructed to make use of the `INTEVAL` and `REVERSEPROP` callbacks of nonlinear handlers. Further, bounds that are implied by the domain of expressions are enforced, if possible, such as the lower bound for arguments of $\log(x)$ or x^p with $p \notin \mathbb{Z}$ are set to a small positive value.

4.2.8 Domain Propagation

As in previous SCIP versions, `cons_nonlinear` implements a feasibility-based bound tightening (FBBT) procedure. For that, interval arithmetic is used to bound the preimage of each constraint function with respect to the constraint sides, i.e., interval over-estimates are computed for

$$\{x \in [\underline{x}, \overline{x}] : \underline{g}_j \leq g_j(x) \leq \overline{g}_j\}$$

for each $j = 1, \ldots, m$. As it is nontrivial to do so for an arbitrary function, the expression graph and extended formulation ($\text{MINLP}^{\text{dp}}_{\text{ext}}$) are utilized. Recall from Section 4.2.4 that it is assumed that the nonlinear handlers provide methods to compute interval enclosures of $h_i^{\text{dp}}(\cdot)$ and its inverse, see (13)–(15).

On the implementation side, domain propagation consists of one or several forward and backward passes through the expression graph. In the forward pass, interval enclosures of (13) are computed using the current local bounds on variables x. The interval enclosures of (13) are used to update the bounds $\underline{w}_i^{\text{dp}}$, $\overline{w}_i^{\text{dp}}$. In the backward pass, interval enclosures of (14) and (15) are computed and used to update the bounds $\underline{x}_j$, $\overline{x}_j$, $\underline{w}_j$, $\overline{w}_j$. Note that constraint sides are taken into account because, initially, $\underline{w}_i^{\text{dp}} = \underline{g}_i$ and $\overline{w}_i^{\text{dp}} = \overline{g}_i$, $i = 1, \ldots, m$.

To only recalculate intervals that may result in a bound tightening, the constraint handler gets notified when the local bounds on an original variable x_i that appears in a nonlinear constraint is changed. If a bound is tightened, then all nonlinear constraints that contain this variable are marked for propagation. If a bound is relaxed, however, then the previously computed bounds on auxiliary variables $\underline{w}^{\text{dp}}$, $\overline{w}^{\text{dp}}$ are marked as invalid. When the domain propagation routine of the constraint handler is called, the expressions of all constraints that were marked for propagation are processed in a depth-first manner (forward pass). If for some $i = 1, \ldots, m$, the interval enclosure of (13) is not a subset of $[\underline{g}_i, \overline{g}_i]$ (that is, the constraint is not redundant with respect to current variable bounds), then h_i^{dp} is queued for backward propagation. The backward propagation queue is then processed in a breadth-first-order. Each time the interval enclosure of (15) provides a sufficient tightening for $[\underline{w}_j^{\text{dp}}, \overline{w}_j^{\text{dp}}]$, h_j^{dp} is appended to the backward propagation queue. If a bound tightening for some x_j is derived from (14), constraints that contain x_j are marked for propagation again, so that another forward pass may start after the current backward pass.

The following mentions a few more subtleties.

Auxiliary Variables in ($\text{MINLP}^{\text{lp}}_{\text{ext}}$) Recall that the `DETECT` callback of a nonlinear handler can request bound updates for auxiliary variables in ($\text{MINLP}^{\text{lp}}_{\text{ext}}$), see Section 4.2.5. Thus, if h_i^{dp} is not only associated with w_i^{dp} but also an auxiliary variable $w_{i'}^{\text{lp}}$, then the bounds on $w_{i'}^{\text{lp}}$ are tightened, too.

Reducing Side Effects The bounds computed in a backward pass are stored separately from those computed by the forward pass. That is, tightened bounds on w^{dp} are not immediately used to compute the bounds on functions that use w^{dp}. Instead, a bound tightening on an auxiliary variable in the backward pass first has to result in a bound change on an original variable x, which should then result in tighter bounds computed by the forward pass. A reason for this implementation detail is that it is tried to reduce side-effects from the backward propagation in a node of the branch-and-bound tree on the domain propagation in another node of the tree. With the current implementation, the domain propagation in a node only depends on the bounds of SCIP variables x and w^{lp}, but not bounds on w^{dp} that were computed by backward propagation in a different part of the tree.

Integrality Integrality information on expressions is taken into account to tighten intervals with fractional bounds to integral values.

Handling Rounding Errors in Variable Bounds and Constraint Sides While the domain propagation in the constraint handler and expression and nonlinear handlers are implemented by using interval arithmetics with outward-rounding, this is not the case for many other parts of SCIP. For this reason, variable bounds are relaxed by a small amount when entering the forward pass. Since these small relaxations result in overestimates for the intervals of all following computations, several cases deserve a special treatment:

- By default, a bound b is relaxed by $10^{-9}\max(1, |b|)$.
- If, however, the domain width is small, but the bound itself is large, then relaxing by $10^{-9}|b|$ can have a large impact. Therefore, bound relaxation is additionally restricted to 10^{-3} times the width of the domain.
- Bounds on integer variables (including implicit integer) are not relaxed.
- Since integral values, especially 0, often have a special meaning, bounds are not relaxed beyond the next integer value.

Constraint sides are relaxed by a small amount, too. Here, an absolute relaxation of 10^{-9} is applied.

Finally, also when updating existing bounds in original or auxiliary variables with newly computed ones, the latter are slightly relaxed if the new interval has a nonzero distance of less than `numerics/epsilon`=10^{-9} to the existing domain. That is, instead of concluding infeasibility for the current subproblem, the variable is fixed to the bound that is closest to the new interval.

Special Case: Redundancy Check When checking whether a constraint can be deleted because it is redundant, it needs to be ensured that the constraint is also satisfied for a solution that violates variable bounds by a small amount. Otherwise, a feasibility check for the solution in the original problem can fail. Hence, when doing a forward pass for the redundancy check, bound for all unfixed variables are relaxed by the feasibility tolerance of SCIP, independent of the variable type. Further, constraint sides are relaxed by the feasibility tolerance as well.

Stopping Criterion In the backward pass, only tightenings that do sufficient progress on the bounds of variables w^{dp} and x are usually applied. This is to avoid many rounds of bound tightening that do only little progress. New bounds are considered sufficiently better than previous ones if the variable gets fixed, the relative improvement on a bound is at least `numerics/boundstreps`=5%, or a bound changes sign, i.e., is moved to or beyond zero.

4.2.9 Initialization of Solve and Relaxations

After presolve, SCIP calls the constraint handlers to initialize their data structures for the branch-and-bound process and to initialize the linear and nonlinear relaxations of the problem. For `cons_nonlinear`, the following operations are performed.

Nonlinear Relaxation For each constraint of (MINLP), a simple check for convexity and concavity of function $g_i(x)$ on $[\underline{x}, \overline{x}]$ is done. This uses the `CURVATURE` callback of the expression handlers. The constraint is added to the NLP relaxation of SCIP and the row in the NLP is marked as convex or concave, if possible. This information is picked up by other plugins that work on a convex nonlinear relaxation of the problem, for example, `sepa_convexproj` and `prop_nlobbt`.

Extended Formulations The extended formulations ($\text{MINLP}^{\text{dp}}_{\text{ext}}$) and ($\text{MINLP}^{\text{lp}}_{\text{ext}}$) are setup. That is, the `DETECT` callback of nonlinear handlers are called on the expressions in the nonlinear constraints to identify structure that can be exploited for domain propagation and linear relaxation, see also Section 4.2.5.

Afterwards, the slack- and auxiliary variables w^{lp} are added to SCIP and are marked as relaxation-only [36]. For expressions $h^{\text{lp}}_i(\cdot)$ that were identified to always have an integral value in a feasible solution (see also Section 4.2.7), the type of variable w^{lp}_i is set to be implicitly integer instead of continuous.

Finally, bounds $\underline{w}^{\text{lp}}, \overline{w}^{\text{lp}}$ are tightened by running a specialized variant of domain propagation (see Section 4.2.8). In this variant, backward propagation is called for all functions $h^{\text{dp}}_i(\cdot)$, $i = 1, \ldots, m^{\text{dp}}$. This is to ensure that domain information that can not be inferred from bounds on the original variables x is stored in the variable bounds $\underline{w}^{\text{lp}}, \overline{w}^{\text{lp}}$. For example, for $\log(w^{\text{lp}}_1)$ with $w^{\text{lp}}_1 = xy$ and $x, y \in [-1, 1]$, the bound $w^{\text{lp}}_1 > 0$ is implied by the expression itself. However, bounds on x and y cannot be tightened such that $xy \geq 0$ is ensured.

Linear Relaxation An initial linear relaxations of nonlinear constraints is constructed by calling the `INITSEPA` of all nonlinear handlers that participate in ($\text{MINLP}^{\text{lp}}_{\text{ext}}$).

The "default" nonlinear handler computes an initial linear relaxation of (16) by calling the `ESTIMATE` callback of the expression handler. This results in a set of linear under- or overestimators of $h^{\text{lp}}_i(x, w^{\text{lp}}_{i+1}, \ldots, w^{\text{lp}}_{m^{\text{lp}}})$, which are completed to valid hyperplanes by adding auxiliary variable w^{lp}_i.

Collect Square and Bilinear Terms All expressions in ($\text{MINLP}^{\text{lp}}_{\text{ext}}$) that are of the form xy or x^2 (where x and y can be either original or auxiliary variables) are collected in a data structure that is easy to traverse and search. This is used by some plugins that work on bilinear terms (Sections 4.5, 4.9).

4.2.10 Separation

After SCIP solved the LP relaxation for a node of the branch-and-bound tree, it calls the separator callback of the constraint handlers and separators to check whether a cutting plane that separates the current LP solution $(\hat{x}, \hat{w})$ is available. For a constraint $\underline{g}_i \leq g_i(x) \leq \overline{g}_i$ of (MINLP) that is violated by $\hat{x}$, the corresponding extended formulation is checked for separating cutting planes. During separation only "strong" cuts desired, by what cutting planes that are more than just barely violated by $(\hat{x}, \hat{w})$ are meant. The quantification of "more than just barely" is left to the separation algorithm (discussed below and in the following sections).

First, if $h_i^{\text{lp}}(x, w_{i+1}^{\text{lp}}, \ldots, w_{m^{\text{lp}}}^{\text{lp}}) \lesseqgtr_i w_i$ is violated by $(\hat{x}, \hat{w})$, then the nonlinear handlers that registered to contribute to the linear relaxation of this constraint are called. For a nonlinear handler that implements the `ENFO` callback, it is left completely to the nonlinear handler to decide how to separate $(\hat{x}, \hat{w})$ from (16). The callback is also informed that only "strong" cuts are desired and candidates for branching are not collected. If the `ENFO` callback is not implemented, then the `ESTIMATE` callback must be implemented. Thus, a linear under- or overestimator of $h_i^{\text{lp}}(\cdot)$ is requested from the nonlinear handler and completed to a cutting plane. The cutting plane is deemed as "strong" if the estimator is sufficiently close to the value of $h_i^{\text{lp}}(\cdot)$ in $(\hat{x}, \hat{w})$.

Formally, assume that $h_i^{\text{lp}}(\hat{x}, \hat{w}_{i+1}^{\text{lp}}, \ldots, \hat{w}_{m^{\text{lp}}}^{\text{lp}}) > \hat{w}_i^{\text{lp}}$ and that a nonlinear handler provides a linear underestimator $\ell(x, w_{i+1}^{\text{lp}}, \ldots, w_{m^{\text{lp}}}^{\text{lp}})$ of $h_i^{\text{lp}}(\cdot)$ with respect to current local variable bounds. If $\ell(\hat{x}, \hat{w}_{i+1}^{\text{lp}}, \ldots, \hat{w}_{m^{\text{lp}}}^{\text{lp}}) > \hat{w}_i^{\text{lp}}$, then $\ell(x, w_{i+1}^{\text{lp}}, \ldots, w_{m^{\text{lp}}}^{\text{lp}}) \leq w_i^{\text{lp}}$ is a cutting plane that separates $(\hat{x}, \hat{w})$ and is valid for the current branch-and-bound node (and it is valid globally if $\ell(\cdot)$ does not depend on local variable bounds). Further, the cut is regarded as *strong* if

$$\ell(\hat{x}, \hat{w}_{i+1}^{\text{lp}}, \ldots, \hat{w}_{m^{\text{lp}}}^{\text{lp}}) \geq \hat{w}_i^{\text{lp}} + \alpha(h_i^{\text{lp}}(\hat{x}, \hat{w}_{i+1}^{\text{lp}}, \ldots, \hat{w}_{m^{\text{lp}}}^{\text{lp}}) - \hat{w}_i^{\text{lp}}), \tag{17}$$

where α is given by parameter `constraints/nonlinear/weakcutthreshold` and currently set to 0.2. That is, for a strong cut, it is required that it closes at least 20% of the convexification gap. Note that if $h_i^{\text{lp}}(\cdot)$ is convex (and this is also detected by SCIP), then the linear underestimator is typically a linearization of $h_i^{\text{lp}}(\cdot)$ at $(\hat{x}, \hat{w})$ and thus $\ell(\hat{x}, \hat{w}_{i+1}^{\text{lp}}, \ldots, \hat{w}_{m^{\text{lp}}}^{\text{lp}}) = h_i^{\text{lp}}(\hat{x}, \hat{w}_{i+1}^{\text{lp}}, \ldots, \hat{w}_{m^{\text{lp}}}^{\text{lp}})$.

Once $h_i^{\text{lp}}(x, w_{i+1}^{\text{lp}}, \ldots, w_{m^{\text{lp}}}^{\text{lp}}) \lesseqgtr_i w_i$ has been processed, subexpressions $h_{i'}^{\text{lp}}$ of h_i^{lp} are inspected and the nonlinear handler associated with $h_{i'}^{\text{lp}}$ are called for separation or linear under-/overestimation.

Again, some more subtleties are discussed next.

Constraints to Separate Separation is not called for every violated nonlinear constraint of $(\text{MINLP}_{\text{ext}}^{\text{lp}})$. For a subexpression $h_{i'}^{\text{lp}}(\cdot)$ of $h_i^{\text{lp}}(\cdot)$ (including $h_i^{\text{lp}}(\cdot)$ itself), $i \in \{1, \ldots, m\}$, separation is only called if the absolute violation of

$$h_{i'}^{\text{lp}}(x, w_{i+1}^{\text{lp}}, \ldots, w_{m^{\text{lp}}}^{\text{lp}}) \lesseqgtr_i w_i$$

is at least a certain factor of the violation of original constraint $\underline{g}_i \leq g(x) \leq \overline{g}_i$. This factor is controlled by parameter `constraints/nonlinear/enfoauxviolfactor` and currently set to 0.01. This threshold has been added to prevent the separation for constraints whose violation does not contribute significantly to the violation of original constraints. In terms of the first example from Section 4.2.2 ($\log(x)^2 + 2\log(x)y + y^2 \leq 4$), this means that if the violation of $(w_2^{\text{lp}})^2 + 2w_2^{\text{lp}}y + y^2 = w_1^{\text{lp}}$ is very small in comparison to the violation of $\log(x)^2 + 2\log(x)y + y^2 \leq 4$, then separation for the quadratic equation is suspended until the violation of $\log(x) = w_2^{\text{lp}}$ has been sufficiently reduced.

Further, separation is skipped for nonlinear constraints of ($\text{MINLP}^{\text{lp}}_{\text{ext}}$) if their absolute violations is below the feasibility tolerance of SCIP, as no strong cuts are expected in this case.

Cut Cleanup Before a cut is passed to the separation storage of SCIP, its numerical properties are checked and improved, if possible. For a cut $\sum_{j=1}^{k} a_j x_j \leq b$ (x_j refers here to either original or auxiliary variables of ($\text{MINLP}^{\text{lp}}_{\text{ext}}$)) with $|a_j| > |a_{j+1}|$, $j = 1, \ldots, k-1$, $a_k \neq 0$, the following operations are performed:

1. Ensure that the coefficient range $|\frac{a_1}{a_k}|$ is below a certain threshold, which by default is set to 10^7 (parameter `separating/maxcoefratiofacrowprep`). To achieve this, the procedure tries to eliminate either the term $a_1 x_1$ or $a_k x_k$ from the cut by adding the inequality $-a_j x_j \leq -a_j \underline{x}_j$ if $a_j > 0$, or $-a_j x_j \leq -a_j \overline{x}_j$ if $a_j < 0$, for $j = 1$ or $j = k$. Whether the first or the last term is chosen depends on finiteness of variable bounds and the amount that the cut is relaxed in $\hat{x}$ by this operation. Since local variable bounds are used here, a cut that was previously globally valid may now be locally valid only.
2. If the absolute value of the maximal coefficient, $|a_1|$, is below 10^{-4} or above 10^4 (parameter `constraints/nonlinear/strongcutmaxcoef`), then the cut is scaled by a factor 2^p, $p \in \mathbb{Z}$, that suffices to ensure $|a_1| \in [10^{-4}, 10^4]$. Thus, together with the previous criterion, this ensures that the absolute value of all coefficients is within $[10^{-4}, 10^4]$ and aims on sorting out cutting planes whose absolute violation is very small or large due to bad scaling only.
3. Ensure that fractional coefficients are not within $\varepsilon = 10^{-9}$ (`numerics/epsilon`) of an integer value. That is, for a_j with $0 < |a_j - \lfloor a_j \rceil| \leq \varepsilon$, where $\lfloor a_j \rceil$ denotes the integer-rounding of a_j, the cut is relaxed by adding the inequality $(\lfloor a_j \rceil - a_j) x_j \leq (\lfloor a_j \rceil - a_j) \overline{x}_j$ if $\lfloor a_j \rceil > a_j$ or $(\lfloor a_j \rceil - a_j) x_j \leq (\lfloor a_j \rceil - a_j) \underline{x}_j$ if $\lfloor a_j \rceil < a_j$. Due to the use of bound information, a previously globally valid cut may be only locally valid now. The main motivation for this operation is to prevent the replacement of a_j by $\lfloor a_j \rceil$ that would occur when the cut is stored in a `SCIP_ROW`, since that could make the cut invalid.
4. Similar to the previous point, a right-hand side b that is very close to 0 is relaxed. If $b \in [-10^{-9}, 0]$, then b is relaxed to 0. If $b \in (0, 10^{-9}]$, then b is relaxed to $1.1 \cdot 10^{-9}$. This is done to prevent the replacement of $b \in (0, 10^{-9}]$ by 0 that would occur when a `SCIP_ROW` is formed.

If the cleanup failed, for example, because finite bounds were not available to relax the cut, the relaxed cut is no longer violated in $\hat{x}$, or is no longer "strong" ((17) is one way of defining what a "strong" cut is), then it is discarded.

Linearization in Incumbents In the last decades, solvers for convex MINLPs have demonstrated that the choice of the reference point in which to linearize convex nonlinear constraints is essential. While using the solution of the LP relaxation still leads to a convergent algorithm [55], better performance is achieved by using a reference point that is close to or at the boundary of the feasible region [26, 99]. Therefore, also the new implementation of `cons_nonlinear` includes a feature where feasible solutions are used as reference points to generate cutting planes.

That is, whenever a primal heuristic finds a new feasible solution x^*, SCIP iterates through the nonlinear constraints of ($\text{MINLP}^{\text{lp}}_{\text{ext}}$) in reverse order, sets $(w_i^{\text{lp}})^* := h_i^{\text{lp}}(x^*, (w_{i+1}^{\text{lp}})^*, \ldots, (w_{m^{\text{lp}}}^{\text{lp}})^*)$ and calls the `ESTIMATE` callback of the registered nonlinear handler (if it implements this callback) with (x^*, w^*) as reference point. If a globally valid underestimator $\ell(x, w_{i+1}^{\text{lp}}, \ldots, w_{m^{\text{lp}}}^{\text{lp}})$ is returned with $\ell(x^*, (w_{i+1}^{\text{lp}})^*, \ldots, (w_{m^{\text{lp}}}^{\text{lp}})^*) = (w_i^{\text{lp}})^*$ (that is, it is supporting $h_i^{\text{lp}}(\cdot)$ at (x^*, w^*)), then the cut $\ell(x, w_{i+1}^{\text{lp}}, \ldots, w_{m^{\text{lp}}}^{\text{lp}}) \leq w_i^{\text{lp}}$ is added to the cutpool of SCIP. Overestimators are handled analogously. However, since

this feature gave mixed computational results when it was added, it is currently disabled by default (parameter `constraints/nonlinear/linearizeheursol`).

4.2.11 Enforcement

The enforcement callbacks of constraint handlers are the ones where resolving infeasibility of solutions has to be taken most seriously. While domain propagation and separation callbacks are allowed to return empty-handed, the enforcement for nonlinear constraint needs to find some action to enforce violated nonlinear constraints in a given solution point. Especially when points are almost feasible, i.e., when violations in ($\text{MINLP}^{\text{lp}}_{\text{ext}}$) are small (reconsider also the motivating example from Section 4.2.1), enforcing constraints can be difficult and some measures taken may appear desperate.

In summary, the constraint handler attempts to enforce constraints of (MINLP) by separation on ($\text{MINLP}^{\text{lp}}_{\text{ext}}$), domain propagation on ($\text{MINLP}^{\text{dp}}_{\text{ext}}$), or branching on a variable x_i, $i \in \{1, \dots, n\}$.

In the unlikely case that no relaxation has been solved, then the constraint handler is asked to enforce the pseudo-solution (`ENFOPS` callback), that is, a vertex of the variables domain with best objective function value. In this case, domain propagation is called (see Section 4.2.8). If no bound change is found and infeasibility of the node is not concluded, then all variables in all violated nonlinear constraints and with domain width larger than ε (`numerics/epsilon`) are registered as branching candidates. The branching rules of SCIP for external branching candidates will then take care of selecting a variable for branching. If no branching candidate could be found, then it is not clear whether there is no feasible solution left in the current node (though relevant domains are tiny). In this case, the constraint handler instructs SCIP to solve the LP relaxation.

When the constraint handler has to enforce a solution $(\hat{x}, \hat{w})$ of the LP relaxation (`ENFOLP` callback), then the following steps are taken:

1. The violation of the solution in (MINLP) and ($\text{MINLP}^{\text{lp}}_{\text{ext}}$) is analyzed. Let v^g, v^h, and v^b, be the maximal absolute violation of the nonlinear constraints in (MINLP), the nonlinear constraints in ($\text{MINLP}^{\text{lp}}_{\text{ext}}$), and the bounds of variables in nonlinear constraints $(\underline{x}, \overline{x}, \underline{w}, \overline{w})$, respectively. Further, let tol^{feas} be the feasibility tolerance of SCIP (`numerics/feastol`), tol^{lp} be the current primal feasibility tolerance of the LP solver, and ε be the value of `numerics/epsilon`. By default, $\text{tol}^{\text{feas}} = \text{tol}^{\text{lp}} = 10^{-6}$ and $\varepsilon = 10^{-9}$. Thus, if $v^g \leq \text{tol}^{\text{feas}}$, then all nonlinear constraints are satisfied with respect to SCIPs feasibility tolerance and no enforcement is necessary. Further, note that SCIP itself already ensures $v^b \leq \text{tol}^{\text{lp}}$ and $\text{tol}^{\text{lp}} \leq \text{tol}^{\text{feas}}$.
2. If $v^b > v^h$, that is, the violation of variable bounds is larger than violations of the nonlinear constraints in ($\text{MINLP}^{\text{lp}}_{\text{ext}}$), then chances to derive cutting planes from ($\text{MINLP}^{\text{lp}}_{\text{ext}}$) that separate $(\hat{x}, \hat{w})$ are low. This is because methods that work on nonconvex constraints often take variable bounds into account and do not work well when the reference point is outside these bounds. Hence, if $v^b > v^h$ and $\text{tol}^{\text{lp}} > \varepsilon$, then tol^{lp} is reduced to $\max(\varepsilon, v^b/2)$ and a resolve of the LP is triggered.
3. If $v^h < \text{tol}^{\text{lp}}$, that is, violations of the nonlinear constraints in ($\text{MINLP}^{\text{lp}}_{\text{ext}}$) are below the feasibility tolerance of the LP solver, then deriving a valid cut that is violated in the current LP solution by more than tol^{lp} can be very difficult. Therefore, if also $\text{tol}^{\text{lp}} > \varepsilon$, then tol^{lp} is reduced to $\max(\varepsilon, v^h/2)$ and a resolve of the LP is triggered.
4. The separation algorithm from Section 4.2.10 is called with some additional flags that indicate that it is called from the enforcement callback. These additional flags extend the separation algorithm as follows.
   - When the `ENFO` or `ESTIMATE` callbacks of a nonlinear handler are called, then they

are instructed to register variables x_j or w_i^{lp} for branching, if useful. A variable should be registered as a branching candidate if branching on that variable could result in finding tighter cutting planes on the resulting subproblems. Usually, this is the case when a convexification gap was introduced due to convexification of a nonconvex function with respect to the current variable domain. Thus, nonlinear handler that underestimate convex expressions usually do not register branching candidates.

– A forward pass of domain propagation in ($\text{MINLP}_{\text{ext}}^{\text{dp}}$) (see Section 4.2.8) is run to ensure that recent bound tightenings are taken into account.
– Recall that for a violated constraint $\underline{g}_i \leq g_i(x) \leq \overline{g}_i$ with $i \in \{1, \ldots, m\}$, constraint $h_i^{\text{lp}}(x, w_{i+1}^{\text{lp}}, \ldots, w_{m^{\text{lp}}}^{\text{lp}}) \lesseqgtr_i w_i$ and subexpressions of h_i^{lp} are tried for separation. If for none of them a "strong" cut could be found, no branching candidate was registered, and the violation of constraint $\underline{g}_i \leq g_i(x) \leq \overline{g}_i$ is at least $0.5\, v^g$ (parameter `constraints/nonlinear/weakcutminviolfactor`), then separation is repeated without the requirement that cutting planes need to be "strong".
– Dropping the requirement for "strong" cuts has various consequences on the separation algorithm described in Section 4.2.10: The requirement that the absolute violation of constraints of ($\text{MINLP}_{\text{ext}}^{\text{lp}}$) is at least tol^{feas} is dropped (recall again the motivating example from Section 4.2.1 where a solution was feasible with respect to tol^{feas} for ($\text{MINLP}_{\text{ext}}^{\text{lp}}$) but not feasible for (MINLP)).

  Instead of (17), it is now sufficient that the violation of the cutting plane in $(\hat{x}, \hat{w})$ is at least tol^{lp}.

  The cleanup of the cut is modified to take the minimal violation tol^{lp} into account. That is, if the violation is in $[10\varepsilon, \text{tol}^{\text{lp}}]$, then it is scaled up to reach a violation of 10^{-4} (parameter `separating/minefficacy(root)`), if possible[3], or at least tol^{lp}. Step 2 in the original cut cleanup (scale to get coefficients into $[10^{-4}, 10^4]$) is replaced by scaling down the cut to achieve $|a_1| < 10/\text{tol}^{\text{feas}}$, if this is possible without the violation to drop under tol^{lp}.

  Since cuts with violations that are just barely above feasibility tolerances are allowed, it is tried to ensure that floating-point rounding-off errors do not falsify the magnitude of the calculated violation. For that, it is required that the violation of the cut $\sum_j a_j x_j \leq b$ is sufficiently large when compared to the terms of the cut, i.e., at least $2^{-50} \max(|b|, \max_j |a_j \hat{x}_j|)$ is required. The value 50 has been chosen because the mantissa of a floating-point number in double precision has 52 bits.

  The cut cleanup procedure is instructed to record for which variables it has modified coefficients in order to achieve the desired coefficient range or to avoid coefficients to be within ε of an integral value. If the cleanup failed to produce a violated cut, then these variables are registered as branching candidates (auxiliary variables may be mapped onto original variables, though, see Section 4.2.12). The motivation is that since a bound of these variables was used to relax the cut, having a smaller domain may result in less relaxation and thus a higher chance to find a violated cut.

Case Study Most of the "cut cleanup" routines have been added to improve numerical stability on test instances. One of the more peculiar cases is detailed in the following. On instance `ex1252` from MINLPLib, constraint `e4` is originally given as $-6.52(0.00034x_6)^3 - 0.102(0.00034x_6)^2 x_{12} + 7.86 \cdot 10^{-8} x_{12}^2 x_6 + x_3 = 0$ (coefficients have been rounded). After simplification of expressions, this is represented in SCIP as $x_3 - 2.54 \cdot 10^{-10} x_6^3 - 1.17 \cdot 10^{-8} x_6^2 x_{12} + 7.86 \cdot 10^{-8} x_6 x_{12}^2 = 0$. When ($\text{MINLP}_{\text{ext}}^{\text{lp}}$) is constructed, none of the specialized nonlinear handlers detect a structure, so nonlinear handler "default" introduces

[3] See implementation of `SCIPcleanupRowprep()` for more details.

an auxiliary variable for each nonlinear term. The resulting constraint[4] in ($\text{MINLP}^{\text{lp}}_{\text{ext}}$) is $x_3 - 2.54 \cdot 10^{-10} w_{13} - 1.17 \cdot 10^{-8} w_{14} + 7.86 \cdot 10^{-8} w_{16} = w_{12}$. Assume that in a solution the value in the left-hand side is below the one on the right-hand side. Though the constraint is actually linear, enforcing it uses the separation procedures of the nonlinear handler. Therefore, the cut that is generated via the help of the expression handler "sum" is, not surprisingly, $x_3 - 2.54 \cdot 10^{-10} w_{13} - 1.17 \cdot 10^{-8} w_{14} + 7.86 \cdot 10^{-8} w_{16} \geq w_{12}$. This cut is marked as globally valid. Next, the cut cleanup procedure is run and recognizes that the coefficient range is $\approx 10^{10} > 10^7$. It then uses the variable bounds at the current node to eliminate variables from the cut until the coefficient range is sufficiently reduced. Apparently, the least relaxation is necessary if the terms for x_3, w_{12}, and w_{13} are removed. The resulting cut, now only valid for the current node, is $7.86 \cdot 10^{-8} w_{16} - 1.17 \cdot 10^{-8} w_{14} \geq -13.94$, which turns out to be no longer violated by the solution to be separated. Since the relaxation of the cut used the bounds of x_3, w_{12}, and w_{13}, the only choice left to resolve the violation is to tighten these bounds. Therefore, variables x_3 and x_6 (due to $w_{13} = x_6^3$) are registered as branching candidates (in the current implementation, only the left-hand side of constraints in ($\text{MINLP}^{\text{lp}}_{\text{ext}}$) are considered). In a later node, the whole procedure repeats, but since variable bounds are tighter, cut cleanup results in the cut $7.86 \cdot 10^{-8} w_{16} - 1.17 \cdot 10^{-8} w_{14} \geq -12.68$, which has a higher chance to be violated. Eventually the instance can be solved to a gap below 1%, but the challenging numerical properties and the costly way they are currently handled take their toll on the performance.

5. If the separation algorithms were not successful, but branching candidates have been collected, then these candidates are either passed on to the SCIP core as external branching candidates or the branching rules of the nonlinear constraint handler are employed. The latter is currently the default (parameter `constraints/nonlinear/branching/external`) and described in more detail in Section 4.2.12 below.

In most situations, it is either possible to separate an infeasible solution or to find a variable in a nonconvex term such that branching on that variable should reduce the convexification gap, which would allow for a tighter linear relaxation. However, enforcement needs to handle also the less likely situations where neither separation nor branching was successful. This leads to the following (less strategical) attempts.

6. If $v^b > \varepsilon$, then tol^{lp} is reduced to $\max(\varepsilon, v^b/2)$ and a resolve of the LP is triggered. As in Step 2, the hope is that separation methods will work better if the LP solution is less outside the variable bounds.
7. If $v^h > \varepsilon$ and $\text{tol}^{\text{lp}} > \varepsilon$, then tol^{lp} is reduced to $\max(\varepsilon, v^b/2)$ and a resolve of the LP is triggered. The hope here is that i) less tolerance on the feasibility for previously generated cuts may lead to a feasible solution, and ii) more cuts can be added if the minimal required violation is reduced.
8. Domain propagation (Section 4.2.8) is run in the hope that some bound change that hasn't previously been found in the separation-and-propagation loop for the current node is discovered now. This bound change may separate the current LP solution or have an influence on the next separation attempts.
9. Any unfixed variable in violated nonlinear constraints is registered as external branching candidate. SCIP then branches on one of these variable and the hope is that infeasibilities in child nodes will be easier to resolve. Note that when the domain width of a variable is reduced to less than ε, then the variable is treated as if fixed to a single value.

[4] The attentive reader observes that the constraint handler is partially responsible for its own misery here by naively replacing each monomial by an auxiliary variable. Adding an automated scaling for newly introduced variables or being more considerate in the simplification step may help here.

10. If all variables in violated constraints are fixed, then it may be the overestimation of variable bounds that prevented domain propagation to conclude that the current node is infeasible. The node will be cut off and a message issues to the log.

The constraint handler collects statistics on how often it added "weak" cuts, tightened the LP feasibility tolerance (tol^{lp} is reset to tol^{feas} whenever processing of a new node starts), branched on any unfixed variable, etc. The occurrence of such behavior is an indication that SCIP has numerical problems to solve this instance. To see these statistics, enable parameter `table/cons_nonlinear/active`.

Finally, if the constraint handler has to enforce a solution of a relaxation other than the LP (`ENFORELAX`), then almost the same algorithm is run as for enforcing LP solutions. The only differences are that i) tol^{feas} is used instead of tol^{lp} as minimal required cut violation and ii) reduction of tol^{lp} is omitted. Note, that the enforcement of relaxation solutions has not been tested and would probably require some patching up to work reliably.

4.2.12 Branching

The handler for nonlinear constraints now includes its own branching rule to select a variable for branching among a number of candidates. The candidates are variables that usually appear in nonconvex expressions of violated nonlinear constraints and are collected while trying to find a cutting plane that separates a given relaxation solution (Step 4 in the previous section). Branching on such a variable should reduce the gap that is introduced by convexifying the nonconvex expression in both children because this gap is typically proportional to the domain width.

Mapping Constraint Violation onto Variables Within the `ESTIMATE` and `ENFO` callbacks of a nonlinear handler, the handler should register with the constraint handler those variables of ($\text{MINLP}^{\text{lp}}_{\text{ext}}$) where branching could potentially help to produce tighter estimators or cutting planes. With the branching candidates a "violation score" is enclosed, which typically is the relative violation of the nonlinear constraint in ($\text{MINLP}^{\text{lp}}_{\text{ext}}$) that is currently handled,

$$s^v := \frac{|h_i^{\text{lp}}(\hat{x}, \hat{w}_{i+1}^{\text{lp}}, \dots, \hat{w}_{m^{\text{lp}}}^{\text{lp}}) - \hat{w}_i^{\text{lp}}|}{\max(1, |\hat{w}_i^{\text{lp}}|)} \tag{18}$$

This value serves as a proxy for the convexification gap associated with $h_i^{\text{lp}}(\cdot)$.

For each branching candidate, the number of violation scores that have been added, the maximal score, and the sum of scores are stored. If a nonlinear handler registers only one branching candidate for an expression, then the value s^v can be added to the score of that variable immediately. For a multivariate function $h_i^{\text{lp}}(\cdot)$, several candidates may be registered, which requires distributing the s^v onto several variables. Let $x_{i_1}, \dots, x_{i_k}$, $\{i_1, \dots, i_k\} \subseteq \mathcal{N}$, be such a set of variables. Let k_u be the number of unbounded variables in this set,

$$k_u := |\{j \in \{1, \dots, k\} : \underline{x}_{i_j} = -\infty \text{ or } \overline{x}_{i_j} = \infty\}|,$$

If $k_u > 0$, then to each unbounded variable an equal part of the violation score is assigned. That is, variable x_{i_j} is assigned the score

$$\begin{cases} \frac{s^v}{k_u}, & \text{if } \underline{x}_{i_j} = -\infty \text{ or } \overline{x}_{i_j} = \infty, \\ 0, & \text{otherwise.} \end{cases} \tag{19}$$

Hence, only unbounded variables are considered for branching. This is because the computation of a linear outer-approximation of (16) often depends on the presence of

variable bounds. If all variables are bounded, the following variable weights are considered instead:

$$\lambda_j := \begin{cases} \max\left(0.05, \frac{\min(\hat{x}_{i_j} - \underline{x}_{i_j}, \overline{x}_{i_j} - \hat{x}_{i_j})}{\overline{x}_{i_j} - \underline{x}_{i_j}}\right), & \text{if } \underline{x}_{i_j} \neq \overline{x}_{i_j}, \\ 0, & \text{otherwise,} \end{cases} \qquad j = 1, \ldots, k. \tag{20}$$

Value $\lambda_j \in [0.05, 0.5]$ measures the "midness" of the current solution point with respect to the variables domain. Larger shares of the violation score are then assigned to variables that are closer to the middle of the domain:

$$\frac{\lambda_j}{\sum_{j'=1}^{k} \lambda_{j'}} s^v. \tag{21}$$

This choice is inspired by the observation that the convexification gap is typically smallest at the boundary of the domain. Further, since a value close to $\hat{x}_{i_j}$ is typically selected as branching point, this choice prefers variables that lead to children in the branch-and-bound tree with similar domain sizes. The following alternatives to the weights (20) for a bounded unfixed variable x_{i_j} can be chosen (parameter `constraints/nonlinear/branching/violsplit`):

$$\text{uniform: } 1.0$$

$$\text{domain width: } \overline{x}_{i_j} - \underline{x}_{i_j}$$

$$\text{logarithmic scale of domain width: } \begin{cases} 10 \log_{10}(\overline{x}_j - \underline{x}_{i_j}), & \text{if } \overline{x}_j - \underline{x}_{i_j} \geq 10, \\ \frac{1}{-10 \log_{10}(\overline{x}_j - \underline{x}_{i_j})}, & \text{if } \overline{x}_j - \underline{x}_{i_j} \leq 0.1, \\ \overline{x}_j - \underline{x}_{i_j}, & \text{otherwise.} \end{cases}$$

Auxiliary Variables While the choice of notation in the previous section implied that violation scores would only be distributed onto original variables x_i, $i \in \mathcal{N}$, it is clear that the same formulas can be used if some or all variables are auxiliary variables of the extended formulation ($\text{MINLP}^{\text{lp}}_{\text{ext}}$). However, recall that an auxiliary variable w^{lp}_i is essentially just a proxy for a subexpression that is defined with respect to original variables and other auxiliary variables $w^{\text{lp}}_{i'}$, $i' > i$. Due to this construction, branching on original variables x_i is usually preferred, as this tightens not only the bounds on x_i directly but also the bounds on one or several auxiliary variables implicitly (see Section 4.2.8). On the other hand, there may be situations where branching on auxiliary variables could be preferable (after all, such branching could tighten bounds on original variables via domain propagation, too) as it has a more direct effect on the bounds on auxiliary variables. As we have not come up with an intuitive criterion on when to allow branching on auxiliary variables, currently only the minimal depth required for nodes in the branch-and-bound tree to allow branching on auxiliary variables can be specified (parameter `constraints/nonlinear/branching/aux`). The default is to never branch on auxiliary variables, though. Therefore, when a nonlinear handler registers a set of variables and a violation score for branching, each auxiliary variable w^{lp}_i in this set is replaced by the variables that are appear in $h^{\text{lp}}_i(x, w^{\text{lp}}_{i+1}, \ldots, w^{\text{lp}}_{m^{\text{lp}}})$. This is repeated until only original variables are left. The violation score is then distributed among this set of original variables.

If during enforcement, separation failed to find a cut (recall Step 4 in Section 4.2.11), all variables with an assigned violation score are collected by default. Optionally, only candidates from constraints which violation is a certain factor of v^g are considered for branching. However, this factor is by default 0 (parameter `constraints/nonlinear/branching/highviolfactor`).

Branching Candidate Scores Let $x_{i_1}, \ldots, x_{i_k}$, $\{i_1, \ldots, i_k\} \subseteq \mathcal{N}$, be the set of branching candidates. With each candidate, up to five different scores are associated.

The *violation score* was already introduced in the previous paragraph. When for a variable several violation scores have been added, then currently the sum of these values is used:

$$s_j^v \coloneqq \text{sum of violations scores (19) or (21) assigned to } x_{i_j}, \quad j = 1, \ldots, k.$$

Alternatively, also the average or maximum can be used (parameter `constraints/nonlinear/branching/scoreagg`). Summing up has the effect that variables appearing in several violated constraints are likely to be assigned a larger violation score.

The *pseudo-costs score* mimics the adaptation of pseudo costs to spatial branching as it was introduced by Belotti et al. [12] and is implemented also in the SCIP core and `pscost` branching rule plugin. Pseudo costs associated with a variable x_i are estimates on the change in the dual bound that is provided by the LP relaxation and that result from branching on variable x_i. For brevity, only a simplified explanation of the calculation of pseudo costs and pseudo-cost branching scores for continuous variables and the default setting for `branching/lpgainnormalize` is presented here. Assume that the LP relaxation has been solved in a node of the branch-and-bound tree after branching on variable x_i at branching point $\tilde{x}_i$. The pseudo costs ψ_i^+ and ψ_i^- store the average change in the LP objective function value normalized by the change in the domain width of x_i. That is, if Δ is the absolute change in the LP relaxations objective function value in the node $x_i \leq \tilde{x}_i$ with respect to the parent node, then value $\Delta/(\tilde{x}_i - \underline{x}_i)$ is used to update the pseudo cost ψ_i^+. Alternatively, for the node $x_i \geq \tilde{x}_i$, ψ_i^- is updated with $\Delta/(\overline{x}_i - \tilde{x}_i)$.

The pseudo-cost score is a prediction of the dual bound gain that can be expected from branching on variable x_{i_j}, $j \in \{1, \ldots, k\}$. Let $\tilde{x}_{i_j} \in (\underline{x}_{i_j}, \overline{x}_{i_j})$ be the branching point (see also end of this section) that would be chosen if branching on x_{i_j}. Then the quantities $\psi_{i_j}^+(\tilde{x}_{i_j} - \underline{x}_{i_j})$ and $\psi_{i_j}^-(\overline{x}_{i_j} - \tilde{x}_{i_j})$ are used to define the pseudo-cost branching score:

$$s_j^p \coloneqq \begin{cases} \text{n/a}, & \text{if } \underline{x}_{i_j} = -\infty \text{ or } \overline{x}_{i_j} = \infty, \text{ otherwise} \\ \psi_{i_j}^+(\tilde{x}_{i_j} - \underline{x}_{i_j}) \cdot \psi_{i_j}^-(\overline{x}_{i_j} - \tilde{x}_{i_j}), & \text{if both } \psi_{i_j}^+ \text{ and } \psi_{i_j}^- \text{ are deemed reliable,} \\ \psi_{i_j}^+(\tilde{x}_{i_j} - \underline{x}_{i_j}), & \text{if only } \psi_{i_j}^+ \text{ is deemed reliable,} \\ \psi_{i_j}^-(\overline{x}_{i_j} - \tilde{x}_{i_j}), & \text{if only } \psi_{i_j}^- \text{ is deemed reliable,} \\ \text{n/a}, & \text{otherwise} \end{cases}$$

A value $\psi_{i_j}^+/\psi_{i_j}^-$ is deemed reliable if it has been updated at least twice (`constraints/nonlinear/branching/pscostreliable`). The pseudo-cost score is not computed for problems with constant objective function ($c = 0$ in (MINLP)).

The *domain score* aims on giving preference to variables with a domain that is not very large or very small. The motivation is that relatively large domains may require many branching operations until their domain is small enough to allow for a useful linear relaxation and branching on relatively small domains may not reduce the convexification gap considerably anymore. The domain score is therefore largest for domains of width 1 and slowly decreases for larger and smaller domains:

$$s_j^b \coloneqq \begin{cases} \log_{10}(2 \cdot 10^{20}/(\overline{x}_{i_j} - \underline{x}_{i_j})), & \text{if } \overline{x}_{i_j} - \underline{x}_{i_j} \geq 1, \\ \log_{10}(2 \cdot 10^{20} \max(\varepsilon, \overline{x}_{i_j} - \underline{x}_{i_j})), & \text{otherwise,} \end{cases}$$

The appearance of 10^{20} in this formula is due to the implicit bound of 10^{20} (`numerics/infinity`) that SCIP applies to unbounded variables. Thus, in this formula, $\underline{x}_{i_j}$ and $\overline{x}_{i_j}$ should be understood as $\pm 10^{20}$ if at infinity.

The *integrality score* aims on giving preference to variables that are of integral type because the domains of integer branching variables in child nodes does not overlap.

Further, binary variables are preferred over integer variables as branching on a binary variable will fix it in both children. The score is defined as

$$s_j^i := \begin{cases} 1.0, & \text{if } i_j \in \mathcal{I}, \underline{x}_{i_j} = 0, \overline{x}_{i_j} = 1, \\ 0.1, & \text{if } i_j \in \mathcal{I}, \underline{x}_{i_j} \neq 0 \text{ or } \overline{x}_{i_j} \neq 1, \\ 0.01, & \text{if } i_j \in \mathcal{N} \setminus \mathcal{I}, x_{i_j} \text{ has been marked to be implicitly integer}, \\ 0.0, & \text{otherwise.} \end{cases}$$

Finally, the *dual score* is a coarse idea that tries to evaluate the importance of violation scores (18) from the perspective of the dual bound that the LP relaxation provides. Assume that for a constraint $h_i^{\text{lp}}(x, w_{i+1}^{\text{lp}}, \ldots, w_{m^{\text{lp}}}^{\text{lp}}) \leq w_i^{\text{lp}}$ of $(\text{MINLP}_{\text{ext}}^{\text{lp}})$ a cut $\ell(x, w_{i+1}^{\text{lp}}, \ldots, w_{m^{\text{lp}}}^{\text{lp}}) \leq w_i^{\text{lp}}$, where $\ell(\cdot)$ is a linear underestimator of $h_i^{\text{lp}}(\cdot)$, was added to the LP. If μ denotes the dual variable associated with this cut in the LP, then this cut contributes $\mu(\ell(x, w_{i+1}^{\text{lp}}, \ldots, w_{m^{\text{lp}}}^{\text{lp}}) - w_i^{\text{lp}})$ to the Lagrangian function of the LP relaxation. If instead of the cut the function $h_i^{\text{lp}}(\cdot)$ could have been used in the LP, then this would change the value of the Lagrangian function by $\mu(h_i^{\text{lp}}(x, w_{i+1}^{\text{lp}}, \ldots, w_{m^{\text{lp}}}^{\text{lp}}) - \ell(x, w_{i+1}^{\text{lp}}, \ldots, w_{m^{\text{lp}}}^{\text{lp}}))$. Therefore, this product of dual variable and convexification gap is used to evaluate the importance that the linear relaxation of this nonlinear constraint has on the dual bound that is provided by the LP.

In the current experimental implementation, the convexification gap

$$|h_i^{\text{lp}}(\hat{\hat{x}}, \hat{\hat{w}}_{i+1}^{\text{lp}}, \ldots, \hat{\hat{w}}_{m^{\text{lp}}}^{\text{lp}}) - \ell(\hat{\hat{x}}, \hat{\hat{w}}_{i+1}^{\text{lp}}, \ldots, \hat{\hat{w}}_{m^{\text{lp}}}^{\text{lp}})|$$

in the LP solution $(\hat{\hat{x}}, \hat{\hat{w}})$ at the time the cut is generated is stored together with the cut. (The use of the absolute value is to accommodate overestimators from the case where $\overset{\leq}{\geq} i$ is $\geq$). To compute the dual score s_j^d of a variable x_{i_j}, for all rows in the LP that contain x_{i_j} and that were generated from a nonlinear constraint of $(\text{MINLP}_{\text{ext}}^{\text{lp}})$, the quantities $|\hat{\mu}(h_i^{\text{lp}}(\hat{\hat{x}}, \hat{\hat{w}}_{i+1}^{\text{lp}}, \ldots, \hat{\hat{w}}_{m^{\text{lp}}}^{\text{lp}}) - \ell(\hat{\hat{x}}, \hat{\hat{w}}_{i+1}^{\text{lp}}, \ldots, \hat{\hat{w}}_{m^{\text{lp}}}^{\text{lp}}))|$ are added. Here, $\hat{\mu}$ refers to the dual value of the cut in the current LP solution. The current implementation has a number of disadvantages that will need to be addressed before the dual score could be usable by default. For example, it would obviously be better to use the convexification gap in the current LP solution instead of $\hat{\hat{x}}$. Further, cuts may be defined in terms of auxiliary variables, but branching is done on original variables only. Thus, the replacement of auxiliary variables by original variables (see paragraph "Auxiliary Variables" above) would need to be considered here as well.

In a final step, the scores s_j^v, s_j^p, s_j^b, s_j^i, s_j^d are aggregated into a single score for each variable. For that, weights γ^v, γ^p, γ^b, γ^i, γ^d are used, which can be set by parameters `constraints/nonlinear/branching/*weight` and default to $\gamma^v = 1.0$, $\gamma^p = 1.0$, $\gamma^b = 0$, $\gamma^i = 0.5$, $\gamma^d = 0$. Since the scores can be of different magnitudes, they are scaled by the maximal score in each category. Thus, let $s_{\max}^v := \max_{j=1,\ldots,k} s_j^v$ and similar for $s_{\max}^p$, $s_{\max}^b$, $s_{\max}^i$, $s_{\max}^d$. Further, the case that pseudo-cost scores may not be available for each variable needs to be considered. Therefore, for a variable where pseudo-cost scores are available, the final score is computed as

$$s_j^f := \frac{\gamma^v \frac{s_j^v}{s_{\max}^v} + \gamma^p \frac{s_j^p}{s_{\max}^p} + \gamma^b \frac{s_j^b}{s_{\max}^b} + \gamma^i \frac{s_j^i}{s_{\max}^i} + \gamma^d \frac{s_j^d}{s_{\max}^d}}{\gamma^v + \gamma^p + \gamma^b + \gamma^i + \gamma^d}.$$

If a pseudo-cost score is not available, then the other scores are magnified:

$$s_j^f := \frac{\gamma^v \frac{s_j^v}{s_{\max}^v} + \gamma^b \frac{s_j^b}{s_{\max}^b} + \gamma^i \frac{s_j^i}{s_{\max}^i} + \gamma^d \frac{s_j^d}{s_{\max}^d}}{\gamma^v + \gamma^b + \gamma^i + \gamma^d}.$$

Branching Variable and Coordinate Since the variable scores are rather a heuristic guideline than a clear indication which variable is "best", the code chooses from all variable with final score at least $0.9 \max_{j=1,\dots,k} s_j^f$ (`constraints/nonlinear/branching/highscorefactor`) uniformly at random. This allows to exploits performance variability due to branching decisions by changing the seed for the random number generator (`randomization/randomseedshift`).

The branching point selection rule has not been changed since the last SCIP release. For a bounded variable x_j, a value $\tilde{x}_j$ between $\hat{x}_j$ and $\frac{1}{2}(\underline{x}_j + \overline{x}_j)$ is chosen, see also Section 4.4.5 of the SCIP Optimization Suite 7.0 release report [36]. Two child nodes are created, one with $x_j \leq \tilde{x}_j$ and another with $x_j \geq \tilde{x}_j$, if $j \notin \mathcal{I}$. For $j \in \mathcal{I}$, domains are ensured to be disjoint ($x_j \leq \lfloor x_j \rfloor$, $x_j \geq \lfloor x_j \rfloor + 1$).

4.3 Nonlinear Handler for Quadratic Expressions

The quadratic nonlinear handler detects quadratic expressions, provides specialized domain propagation, and generates intersection cuts.

4.3.1 Detection of Quadratic Expressions

An expression in a constraint of ($\text{MINLP}_{\text{ext}}^{\text{dp}}$) or ($\text{MINLP}_{\text{ext}}^{\text{lp}}$) is recognized as quadratic if it is a sum of terms where at least one term is either a product expression of two expressions or a power expression with exponent 2. Formally, the detection routine checks whether the expression can be written as

$$q(y) = \sum_{i=1}^{k} q_i(y) \quad \text{with} \quad q_i(y) = a_i y_i^2 + c_i y_i + \sum_{j \in P_i} b_{i,j} y_i y_j \tag{22}$$

where y_i is either an original variable (x) or another expression, $a_i, c_i \in \mathbb{R}$, $b_{i,j} \in \mathbb{R} \setminus \{0\}$, $j \in P_i \Rightarrow i \notin P_j$ for all $j \in P_i$, $P_i \subset \{1, \dots, k\}$, $i = 1, \dots, k$. A bilinear term $y_i y_j$ is associated with y_i (i.e., $j \in P_i$) if y_i appears more often in $q(y)$ than y_j. This is a heuristic choice that should be beneficial for domain propagation. In case of a tie, the order of expressions (see Section 4.1) is used as tie-breaker. If $q(y)$ is linear in y or consists of one power or product term only, then detection is aborted.

After a quadratic structure (22) has been established, the construction of the extended formulations for ($\text{MINLP}_{\text{ext}}^{\text{dp}}$) and ($\text{MINLP}_{\text{ext}}^{\text{lp}}$) can differ.

For domain propagation (($\text{MINLP}_{\text{ext}}^{\text{dp}}$)), the nonlinear handler checks further whether the quadratic expression is *propagable*, by what is meant that the termwise domain propagation does not necessarily yield the best possible results due to suffering from the so-called *dependency problem* of interval arithmetics. Specifically, (22) is propagable if at least one argument (y_i) appears at least twice. For instance, $x^2 + y^2$ is not propagable, but $x^2 + x$ is. Only if (22) is propagable, the nonlinear handler registers itself as responsible for domain propagation. Otherwise, the default nonlinear and the expression handlers for sum, product, and power will take care of a termwise propagation of domain.

To construct ($\text{MINLP}_{\text{ext}}^{\text{dp}}$), the nonlinear handler requests an auxiliary variable (w^{dp}) for any y_i that is an expression and not yet a variable, but with two notable exceptions. If a variable y_i appears only in a square term of (22) ($a_i \neq 0$, $c_i = 0$, $i \notin P_{i'}$ for all $i' = 1, \dots, k$), then an auxiliary variable is introduced for y_i^2 instead of y_i. Similarly, if two variable y_i and y_j appear only as one bilinear term $y_i y_j$ ($a_i = 0$, $a_j = 0$, $c_i = 0$, $c_j = 0$, $P_i = \{j\}$ or $P_j = \{i\}$), then an auxiliary variable is introduced for $y_i y_j$ instead of y_i and y_j. That is, a non-propagable part of (22) is split off and treated as if linear since this part does not suffer from the dependency-problem and sometimes better domain propagation routines are available for the single terms y_i^2 or $y_i y_j$ (for an example see the

bilinear nonlinear handler described in Section 4.5). For an example, consider $xy + z^2 + z$, which is propagable because z appears twice. However, for ($\text{MINLP}^{\text{dp}}_{\text{ext}}$), the reformulation $w + z^2 + z$, $w = xy$, is applied. The quadratic nonlinear handler then handles domain propagation for $w + z^2 + z$, while either the default or the bilinear nonlinear handler handle domain propagation for $w = xy$. An additional advantage of this division of work is that for other expression where xy appears, variable w and its domain information can be reused.

For separation (($\text{MINLP}^{\text{lp}}_{\text{ext}}$)), the nonlinear handler registers itself for participation if intersection cuts are enabled (`nlhdlr/quadratic/useintersectioncuts`, currently disabled by default), no other nonlinear handler (for example the SOC nonlinear handler) handles separation yet, and the corresponding constraint in ($\text{MINLP}^{\text{lp}}_{\text{ext}}$) is nonconvex. To decide the latter, the eigenvalues and eigenvectors of the quadratic coefficients matrix (defined by a_i and $b_{i,j}$ of (22)) are calculated via LAPACK and stored for later use. To construct ($\text{MINLP}^{\text{lp}}_{\text{ext}}$), the nonlinear handler requests an auxiliary variable (w^{lp}) for any y_i that is an expression and not yet a variable. Thus, even when the quadratic nonlinear handler participates in both domain propagation and separation, the created extended formulations may differ if parts of (22) are not propagable. This flexibility is a feature of the current design.

A nonlinear handler can choose whether it solely wants to be responsible for domain propagation or separation, or only wants to participate in addition to other routines. The separation by the quadratic nonlinear handler is such a case, i.e., the nonlinear handler informs to the constraint handler that other possible nonlinear handlers should also be requested for separation. Currently, this means that the default and bilinear nonlinear handler will become active, auxiliary variables will be introduced for each square and bilinear term, and corresponding under- and overestimators will be computed by these routines if an intersection cut was not generated. These nonlinear handlers are also the only ones that register branching candidates. For intersection cuts, bound information is not used explicitly by default and the quadratic nonlinear handler does not register variables for branching.

4.3.2 Propagation of Quadratic Expressions

The goal of domain propagation is to use existing bounds on y and $q(y)$ in (22) to derive possibly tighter bounds on $q(y)$ and y, respectively. The implementation is similar to the one of `cons_quadratic` in SCIP 7 and before [115], but backward propagation has been extended. For simplicity, the special treatment for some square or bilinear terms as mentioned in the previous section is disregarded here.

Forward Propagation (`INTEVAL`, (13)) Here, the goal is to propagate bounds on y to the quadratic expression $q(y)$. To obtain the best bounds, it would be necessary to maximize/minimize $q(y)$ for $y \in [\underline{y}, \overline{y}]$. Since this is too expensive in general, bounds are overestimated. For that, each quadratic term $q_i(y)$, $i = 1, \ldots, k$, is considered separately. If $q_i(y)$ is not univariate ($P_i \neq \emptyset$), then each y_j, $j \in P_i$, is replaced by its current bounds and the min/max of $a_i y_i^2 + (c_i + \sum_{j \in P_i} b_{i,j}[\underline{y}_j, \overline{y}_j])y_i$ are calculated [24].

Backward Propagation (`REVERSEPROP`, (14), (15)) The goal of backward propagation is to derive bounds on each y_i from given bounds $[\underline{q}, \overline{q}]$ on $q(y)$ and current bounds on y. Similar to forward propagation, the tightest bounds can be derived by maximizing/minimizing each y_i w.r.t. $y \in [\underline{y}, \overline{y}]$ and $q(y) \in [\underline{q}, \overline{q}]$. Since this is again too expensive to do in general, bounds are overestimated again.

Let $[\underline{q}_i, \overline{q}_i]$, $i = 1, \ldots, k$, be bounds on each $q_i(y)$ for $y \in [\underline{y}, \overline{y}]$. These are computed in forward propagation. Similar to forward propagation, bounds for each y_i can be

computed by reduction to and solving of a univariate quadratic interval equation [24]:

$$a_i y_i^2 + (c_i + \sum_{j \in P_i} b_{i,j}[\underline{y}_j, \overline{y}_j])y_i \in [\underline{q}, \overline{q}] - \sum_{i'=1, i' \neq i}^{k} [\underline{q}_{i'}, \overline{q}_{i'}]. \tag{23}$$

A downside of this approach is that bounds for variables that appear less often may not be deduced. For example, consider $y_1^2 + y_1 y_2 + y_1 y_3 + y_2 y_3 + y_3$. As y_2 has less appearances than y_1 and y_3, this quadratic gets partitioned into $q_1(y) = y_1^2 + y_1 y_2 + y_1 y_3$, $q_2(y) = 0$, and $q_3(y) = (y_3 + y_3 y_2)$. Therefore, no bounds are computed for y_2 in backward propagation. The quadratic constraint handler of SCIP 7 handled the case of $q_i \equiv 0$ in certain situations where y_i appeared in only one bilinear term. For SCIP 8, this has been generalized. In the example, a bound on y_2 is obtained by rewriting as $y_2 + y_3 \in ([\underline{q}, \overline{q}] - [\underline{q}_3, \overline{q}_3])/y_1 - y_1$, finding the min/max of the function on the right-hand side, and using this interval for backward propagation on $y_2 + y_3$. In general, after solving (23), the quadratic equation $q(y) \in [\underline{q}, \overline{q}]$ is interpreted as

$$c_i + \sum_{j \in P_i} b_{i,j} y_j \in \frac{1}{y_i} \left([\underline{q}, \overline{q}] - \sum_{i'=1, i' \neq i}^{k} [\underline{q}_{i'}, \overline{q}_{i'}] \right) - a_i y_i,$$

the min/max of the univariate interval function on the right-hand side are calculated, and the resulting interval is used for backward propagation on $c_i + \sum_{j \in P_i} b_{i,j} y_j$.

4.3.3 Intersection Cuts for Quadratic Constraints

For separation, assume the constraint of ($\text{MINLP}_{\text{ext}}^{\text{lp}}$) is $q(y) \leq w$ with $q(y)$ as in (22) and w an auxiliary variable of ($\text{MINLP}_{\text{ext}}^{\text{lp}}$). Further, assume that $q(y)$ is nonconvex ($q(y)$ being convex is handled by the nonlinear handler for convex expressions, see Section 4.6). The quadratic nonlinear handler implements the separation of intersection cuts [111, 10, 41] for the set $S := \{(y, w) \in \mathbb{R}^k : q(y) \leq w\}$ that is defined by this constraint.

Let $(\hat{y}, \hat{w})$ be a basic feasible LP solution violating $q(y) \leq w$. First, a convex inequality $g(y, w) < 0$ is build that is satisfied by $(\hat{y}, \hat{w})$, but by no point of S. This defines a so-called *S-free set* $C = \{(y, w) \in \mathbb{R}^{k+1} : g(y, w) \leq 0\}$, that is, a convex set with $(\hat{y}, \hat{w}) \in \text{int}(C)$ containing no point of S in its interior. The quality of the resulting cut highly depends on which S-free set is used. The tightest possible intersection cuts are obtained by using *maximal* S-free sets as proposed by Muñoz and Serrano [81].

By using the conic relaxation K of the LP-feasible region defined by the nonbasic variables at $(\hat{y}, \hat{w})$, the intersection points between the extreme rays of K and the boundary of C are computed. The intersection cut is then defined by the hyperplane going through these points and successfully separates $(\hat{x}, \hat{w})$ and S. Adding this cut to the LP relaxation excludes the violating point $(\hat{x}, \hat{w})$ from the LP-feasible region and thus enforces the quadratic constraint $q(y) \leq w$. To obtain even better cuts, there is also a strengthening procedure implemented that uses the idea of negative edge extension of the cone K [42]. A detailed description of how the (strengthened) intersection cuts are implemented can be found in the paper by Chmiela et al. [21].

4.4 Nonlinear Handler for Second-Order Cones

The nonlinear handler for second-order cone (SOC) structures replaces and extends the previous constraint handler for second-order cone constraints. It detects second-order cone constraints in the original or extended formulation and provides separation by means of a disaggregated cone reformulation.

4.4.1 Detection

Given a constraint $h_i^{\text{lp}}(x) \leq w_i^{\text{lp}}$ ($\lesseqgtr_i$ being $\geq$ is handled similarly) of the extended formulation ($\text{MINLP}_{\text{ext}}^{\text{lp}}$), the nonlinear handler checks for four different structures. In some cases, it distinguishes between constraints that are copies of an original constraint with slack variable w_i^{lp} added, that is, $i \leq m$, and constraints that only exist in the extended formulation due to the introduction of auxiliary variables, i.e., $i > m$. Further, binary variables are treated as if they were squared, since this increases the likelihood of finding a SOC-structure.

Euclidian Norm If $i > m$, it is checked whether $h_i^{\text{lp}}(x)$ has the form

$$\sqrt{\sum_{j=1}^{k}(a_j y_j^2 + b_j y_j) + c} \tag{24}$$

for some coefficients $a_j, b_j, c \in \mathbb{R}$, $a_j > 0$, and where y_j is either an original variable (x) or some subexpression of $h_i^{\text{lp}}(\cdot)$, $j = 1, \ldots, k$, for some $k \geq 2$. Rewriting (24) reveals the constraint

$$\sqrt{\sum_{j=1}^{k}\left(\left(\sqrt{a_j}y_j + \frac{b_j}{2\sqrt{a_j}}\right)^2 - \frac{b_j^2}{4a_j}\right) + c} \leq w_i^{\text{lp}}. \tag{25}$$

If $c - \sum_{j=1}^{k} \frac{b_j^2}{4a_j} \geq 0$, then (25) has SOC-structure. Thus, the nonlinear handlers requests auxiliary variables for each y_j, $j = 1, \ldots, k$, and declares that it will provide separation. In a future version, any positive-semidefinite quadratic expression should be allowed for the argument of $\sqrt{\cdot}$ in (24).

If $i \leq m$, then w_i^{lp} is just a slack-variable and the constraint is equivalent to $\sqrt{\sum_{j=1}^{k}(a_j y_j^2 + b_j y_j) + c} \leq \overline{g}_i$. In that case, the nonlinear handler does not get active. Assuming $a_j > 0$ again, this will result in the extended formulation $\sqrt{w_0} \leq \overline{g}_i$, $\sum_{j=1}^{k}(a_j w_j + b_j y_j + c) \leq w_0$, $y_j^2 \leq w_j$, $j = 1, \ldots, k$, where $w_0, \ldots, w_k$ are new auxiliary variables. We believe that separation for this formulation will be more efficient than for (25) (constraint $\sqrt{w_0} \leq \overline{g}_i$ is easily enforced by domain propagation).

Simple Quadratics Check whether $h_i^{\text{lp}}(x)$ has the form

$$\sum_{j=1}^{k}(a_j y_j^2) - a_{k+1} y_{k+1}^2 + c$$

where $a_j, c \in \mathbb{R}$, $a_j > 0$, and y_j is either an original variable (x) or some subexpression of $h_i^{\text{lp}}(\cdot)$, $j = 1, \ldots, k+1$, for some $k \geq 1$. The relaxed constraint

$$\sum_{j=1}^{k}(a_j y_j^2) - a_{k+1} y_{k+1}^2 + c \leq \overline{w}_i^{\text{lp}} \tag{26}$$

has SOC-structure if $c - \overline{w}_i^{\text{lp}} \geq 0$. Thus, in this case the nonlinear handler requests auxiliary variables for each y_j, $j = 1, \ldots, k+1$, and declares that it will provide separation.

If $i \leq m$, then the replacement of the slack variable w_i^{lp} by $\overline{w}_i^{\text{lp}}$ will not be problematic since it is sufficient to enforce the original constraint $g_i(x) \leq \overline{g}_i$ (recall $h_i^{\text{lp}} = g_i$, $\overline{w}_i^{\text{lp}} = \overline{g}_i$ initially). However, if $i > m$, then relaxing w_i^{lp} to $\overline{w}_i^{\text{lp}}$ could mean that infeasibility in (MINLP) cannot be resolved by enforcing (26). Therefore, if $i > m$, then the nonlinear

handler indicates to the constraint handler that separation should be requested from other nonlinear handlers as well. In the current configuration, this introduces auxiliary variables for each square term in (26) by the default nonlinear handler. The same distinction into $i \leq m$ and $i > m$ applies to the following two structures.

Simple Quadratics (Rotated SOC Variant) Check whether $h_i^{\text{lp}}(x)$ has the form

$$\sum_{j=1}^{k}(a_j y_j^2) - a_{k+1}y_{k+1}y_{k+2} + c$$

where $a_j, c \in \mathbb{R}$, $a_j > 0$, and y_j is either an original variable (x) or some subexpression of $h_i^{\text{lp}}(\cdot)$, $j = 1, \ldots, k+2$, for some $k \geq 0$. The relaxed constraint

$$\sum_{j=1}^{k}(a_j y_j^2) + \frac{a_{k+1}}{4}(y_{k+1} - y_{k+2})^2 - \frac{a_{k+1}}{4}(y_{k+1} + y_{k+2})^2 + c \leq \overline{w}_i^{\text{lp}}$$

has SOC-structure if $c - \overline{w}_i^{\text{lp}} \geq 0$. Thus, in this case the nonlinear handler requests auxiliary variables for each y_j, $j = 1, \ldots, k+2$, and declares that it will provide separation.

General Quadratics Check whether $h_i^{\text{lp}}(x) \leq \overline{w}_i^{\text{lp}}$ is a quadratic constraint that is SOC-representable. As suggested by Mahajan and Munson [66], this is done by computing an eigenvalue-decomposition of the quadratic coefficients matrix via LAPACK and attempting to rewrite $h_i^{\text{lp}}(x)$ as

$$\sum_{j=1}^{k+1}\lambda_j(v_j^\top y + \beta_j)^2 + c$$

where $\lambda_1, \ldots, \lambda_{k+1} \in \mathbb{R}$ are the nonzero eigenvalues with corresponding eigenvectors $v_1, \ldots, v_{k+1} \in \mathbb{R}^\ell$, $\lambda_j > 0$, $j = 1, \ldots, k$, $\lambda_{k+1} < 0$, $\beta_j, c \in \mathbb{R}$, $j = 1, \ldots, k+1$, $k \geq 1$, and y_j is either an original variable (x) or some subexpression of $h_i^{\text{lp}}(\cdot)$, $j = 1, \ldots, \ell$. If $0 \notin v_{k+1}^\top[\underline{y}, \overline{y}] + \beta_{k+1}$ (thus $\sqrt{(v_{k+1}^\top y + \beta_{k+1})^2} = \pm(v_{k+1}^\top y + \beta_{k+1})$ is linear) and $c - \overline{w}_i^{\text{lp}} \geq 0$, then an SOC-structure has been detected, the nonlinear handler requests auxiliary variables for each y_j, $j = 1, \ldots, k$, and declares that it will provide separation.

4.4.2 Separation

The SOC constraint that has been detected before is stored in the form

$$\sqrt{\sum_{j=1}^{k}(v_j^\top y + \beta_j)^2} \leq v_{k+1}^\top y + \beta_{k+1} \tag{27}$$

with $v_j \in \mathbb{R}^\ell$, $j = 1, \ldots, k+1$, where $y_1, \ldots, y_\ell$ are variables of ($\text{MINLP}_{\text{ext}}^{\text{lp}}$).

Since the left-hand side of (27) is convex, a solution $\hat{y}$ that violates (27) can be separated by linearization of the left-hand side of (27) via a gradient cut:

$$\sqrt{\sum_{j=1}^{k}(v_j^\top \hat{y} + \beta_j)^2} + \sum_{i=1}^{\ell}\frac{\partial}{\partial y_i}\sqrt{\sum_{j=1}^{k}(v_j^\top y + \beta_j)^2}\,(y_i - \hat{y}_i) \leq v_{k+1}^\top y + \beta_{k+1}$$

However, if there are many terms on the left-hand side of (27) (k being large), then it can require many cuts to provide a tight linear relaxation of (27). Thus, as suggested by Vielma et al. [114], a disaggregation of the cone is used if $k \geq 3$:

$$(v_j^\top y + \beta_j)^2 \leq z_j(v_{k+1}^\top y + \beta_{k+1}), \quad j = 1, \ldots, k, \tag{28}$$

$$\sum_{j=1}^{k} z_j \leq v_{k+1}^\top y + \beta_{k+1}, \tag{29}$$

where variables $z_1, \ldots, z_k$ are new variables that are added to SCIP and marked as "relaxation-only". A solution $(\hat{y}, \hat{z})$ that violates (27) needs to violate also (28) for some $j \in \{1, \ldots, k\}$ or (29). The latter is already linear and can be added as cut. If a rotated second-order cone constraint (28) is violated from some j, then it is transformed into the standard form

$$\sqrt{4(v_j^\top y + \beta_j)^2 + (v_{k+1}^\top y + \beta_{k+1} - z_j)^2} \leq v_{k+1}^\top y + \beta_{k+1} + z_j$$

and a gradient cut is constructed by linearization of the left-hand side.

If the constraint handler requests only "strong" cuts (see Section 4.2.10), then gradient cuts are only added when their efficacy is at least 10^{-5} (`nlhdlr/soc/mincutefficacy`). The efficacy is the violation of a cut divided by the Euclidian norm of its coefficient vector.

4.5 Nonlinear Handler for Bilinear Expressions

The bilinear nonlinear handler identifies expressions of the form $y_1 y_2$, where y_1 and y_2 are either non-binary variables of (MINLP$^{\text{lp}}_{\text{ext}}$) or other expressions. For a product $y_1 y_2$, the expressions handler for products already provides linear under- and overestimators and domain propagation that is best possible when considering the bounds $[\underline{y}_1, \overline{y}_1] \times [\underline{y}_2, \overline{y}_2]$ only. The nonlinear handler, however, can exploit linear inequalities over y_1 and y_2 to provide possibly tighter linear estimates and variable bounds. These inequalities are found by projection of the LP relaxation onto variables (y_1, y_2). For more details, see Müller et al. [78].

4.6 Nonlinear Handler for Convex and Concave Expressions

Two nonlinear handlers are available that try to detect convexity or concavity of a given expression $h_i^{\text{lp}}(x)$ and provide appropriate linear under- and overestimators. The naming of the nonlinear handlers may be slightly confusing as the convex nonlinear handler checks for concavity of $h_i^{\text{lp}}(x)$ if overestimators are desired and the concave nonlinear handler checks for convexity if underestimators are desired. After all, the detection algorithms of both nonlinear handlers are similar, so that they are discussed together here. The linear estimators are computed differently, though.

In the following only the underestimating case ($\lesseqgtr_i$ being either $\leq$ or $=$) is considered. The overestimating case is handled analogously. The nonlinear handlers do not contribute to domain propagation so far.

4.6.1 Detection

Assume the constraint handler requests that underestimators of $h_i^{\text{lp}}(x)$ need to be found. The convex nonlinear handler then seeks to find subexpressions of $h_i^{\text{lp}}(x)$ that need to be replaced by auxiliary variables $w_{i+1}^{\text{lp}}, \ldots$ such that the remaining expression $h_i^{\text{lp}}(x, w_{i+1}^{\text{lp}}, \ldots)$

is convex. Similarly, the concave nonlinear handler seeks for $h_i^{\text{lp}}(x, w_{i+1}^{\text{lp}}, \ldots)$ to be concave. In both cases, the detection algorithm can aim for the remaining expression to be as large as possible. This point will be revisited later.

To construct a maximal convex subexpression of $h_i^{\text{lp}}(x)$, the usual convexity and concavity detection rules are inverted and applied on $h_i^{\text{lp}}(x)$ in reverse order. To do so, the expression is traversed in depth-first-search order, starting from the root of $h_i^{\text{lp}}(x)$. With each subexpression the requirement of it being convex and/or concave is associated. For the root, this will be convexity. When a subexpression is considered, it is checked whether the subexpression can have the required curvature. This is done by formulating requirements on convexity/concavity on the children of the subexpression. If there are no conditions under which a subexpression can have the required curvature, then it is marked as to be replaced by an auxiliary variable.

For an example, consider the function $-\sqrt{\exp(x)}\sqrt{y} + \exp(x)$ with $\underline{y} = 0$. First, it will be checked under which conditions on its arguments the sum will be convex. This will create the requirements "$\sqrt{\exp(x)}\sqrt{y}$ must be concave" and "$\exp(x)$ needs to be convex". Checking the former, the special structure $\sqrt{\cdot}\sqrt{\cdot}$ (a product of two power expressions, both exponents being 0.5) may be detected and the requirements "$\exp(x)$ must be concave" and "y must be concave" are created. The check for "$\exp(x)$ must be concave" fails, i.e., there are no conditions on x (other than $\underline{x} = \overline{x}$) such that $\exp(x)$ is concave. Therefore, this appearance of $\exp(x)$ is marked for replacement by an auxiliary variable. The check for "y must be concave" succeeds, since the function $y \mapsto y$ is both convex and concave. The remaining check for "$\exp(x)$ needs to be convex" succeeds under the new condition "x needs to be convex", which is satisfied. Thus, the resulting maximal convex subexpression is $-\sqrt{w}\sqrt{y} + \exp(x)$, where w is a new auxiliary variable and $w \leq \exp(x)$ is added to the extended formulation. As this example has shown, it is possible that several appearances of the same subexpression ($\exp(x)$) are treated differently, depending on what requirements are imposed on the subexpression by its parents.

Four checks whether a subexpression can have a required curvature are currently implemented. These are called in the given order.

Product Composition (`nlhdlr/{convex,concave}/cvxprodcomp`) Check whether a given expression is a product of the form $af(bg(x) + c)g(x)$ with constants a, b, c and repeating subexpression $g(x)$. Considering the second derivative and by using available information on bounds and the `CURVATURE` and `MONOTONICITY` callbacks of the expression handlers, a condition on the curvature of $g(\cdot)$ can be derived that is sufficient for $af(bg(x) + c)g(x)$ to be convex or concave.

Signomial (`nlhdlr/{convex,concave}/cvxsignomial`) Check whether a given expression is a signomial, i.e., a product of power expressions, $c\prod_{j=1}^{k} f_j^{p_j}(x)$ with $c, p_j \in \mathbb{R}$ and subexpressions $f_j(x)$, $j = 1, \ldots, k$. If $\underline{f}_j \geq 0$, then the product is convex if i) $p_j < 0$ and $f_j(x)$ is concave for all $j = 1, \ldots, k$ or ii) $\sum_{j=1}^{k} p_j \geq 1$ and there exists a $j^* \in \{1, \ldots, k\}$ such that $p_j < 0$ and $f_j(x)$ is concave for all $j \neq j^*$ and $f_{j^*}(x)$ is convex. Further, the product is concave if $p_j > 0$ and $f_j(x)$ is concave for all $j = 1, \ldots, k$ and $\sum_{j=1}^{k} p_j \leq 1$. These conditions are proven by Maranas and Floudas [69] and Chen and Huang [20] for the case that each $f_j(x)$ is equal to a variable. If $\overline{f}_j < 0$, then one can adapt by replacing f_j by $-f_j$. If the exponents satisfy the given conditions for convexity or concavity of the product, conditions on convexity and concavity of the subexpressions $f_j(x)$ are formulated.

Quadratics (`nlhdlr/{convex,concave}/cvxquadratic`) Check whether a given expression is quadratic, that is, is of the form $q(y)$ given by (22), where each y_j is either an

original variable or a subexpression. With the same methods as in the nonlinear handler for quadratics, the sign of the eigenvalues of the quadratic coefficients matrix of $q(y)$ can be checked to decide whether $q(y)$ is convex or concave. If $q(y)$ has the desired curvature, then it is required that every y_j is linear.

The check on quadratics is currently disabled for the concave nonlinear handler. It is not clear yet under which conditions it is beneficial to compute underestimators on a multivariate concave $q(y)$ via the methods of the concave nonlinear handler instead of handling each square and bilinear term of $q(y)$ separately.

Expression Handler For an expression $f(g_1(x), \ldots, g_k(x))$, call the `CURVATURE` callback of the expression handler for $f(\cdot)$. If implemented and successful, then it provides convexity or concavity requirements for each $g_j(x)$.

As has been pointed out by Tawarmalani and Sahinidis [110], a tighter linear relaxation of a convex set (in the sense that less cuts are required to achieve the same outer-approximation) can usually be obtained when an extended formulation is used for function composition. For instance, for $f(g(x))$ with both $f(\cdot)$ and $g(\cdot)$ being convex, $f(\cdot)$ being monotonically increasing, and $g(\cdot)$ being nonlinear, it is beneficial to consider the extended formulation $f(w)$, $w \geq g(x)$. This is easily achieved in the detection algorithm by changing the requirement on a subexpression from convex or concave to linear (parameter `nlhdlr/convex/extendedform`). Furthermore, the nonlinear handlers ignore expressions $h_i^{\text{lp}}(\cdot)$ that are a sum with more than one non-constant term (parameters `nlhdlr/{convex,concave}/detectsum`), unless the sum is a quadratic expression with at least one bilinear term, for example, $x^2 + 2xy + y^2$.

For the concave nonlinear handler, however, the observation by Tawarmalani and Sahinidis [110] does not apply. Instead, the number of variables in the expression for which estimators need to be computed can be an issue. Therefore, here auxiliary variables are requested for multivariate linear subexpressions. That is, even though concavity of $\log(x+y+z)$ can be recognized, the extended formulation $\log(w)$, $w \leqq x+y+z$, is used. This way, only one- instead of three-dimensional underestimators need to be calculated.

Finally, if $h_i^{\text{lp}}(x)$ were transformed by the nonlinear handler into $h_i^{\text{lp}}(x, w_{i+1}, \ldots, w_{m^{\text{lp}}})$ such that the corresponding expression has only original or auxiliary variables as children, then the detection of the nonlinear handler is reported as failed (parameter `nlhdlr/{convex,concave}/handletrivial`). Instead, the default nonlinear handler will provide linear estimates via the `ESTIMATE` callback of the expression handlers. We assume that these are more efficient than the generic implementation in the convex and concave nonlinear handlers.

4.6.2 Underestimators for Convex Expressions

For the convex function $h_i^{\text{lp}}(x, w_{i+1}^{\text{lp}}, \ldots, w_{m^{\text{lp}}}^{\text{lp}})$ and reference point $(\hat{x}, \hat{w}^{\text{lp}})$, an underestimator is given by computing a tangent on the graph of $h_i^{\text{lp}}(\cdot)$ at $(\hat{x}, \hat{w}^{\text{lp}})$:

$$h_i^{\text{lp}}(\hat{x}, \hat{w}_{i+1}^{\text{lp}}, \ldots, \hat{w}_{m^{\text{lp}}}^{\text{lp}}) + \nabla h_i^{\text{lp}}(\hat{x}, \hat{w}_{i+1}^{\text{lp}}, \ldots, \hat{w}_{m^{\text{lp}}}^{\text{lp}}) \begin{pmatrix} x - \hat{x} \\ w^{\text{lp}} - \hat{w}^{\text{lp}} \end{pmatrix}.$$

If, however, $h_i^{\text{lp}}(x, w_{i+1}^{\text{lp}}, \ldots, w_{m^{\text{lp}}}^{\text{lp}})$ is univariate, that is, $h_i^{\text{lp}}(x, w_{i+1}^{\text{lp}}, \ldots, w_{m^{\text{lp}}}^{\text{lp}}) = f(y)$ for some variable y, and y is integral, then taking the secant on the graph of $f(y)$ can give a tighter underestimator:

$$f(\lfloor \hat{y} \rfloor) + (f(\lfloor \hat{y} \rfloor + 1) - f(\lfloor \hat{y} \rfloor))\,(y - \lfloor \hat{y} \rfloor).$$

4.6.3 Underestimators for Concave Expressions

To simplify notation, let $f(y)$, $y \in \mathbb{R}^k$, be the concave function $h_i^{\text{lp}}(x, w_{i+1}^{\text{lp}}, \ldots, w_{m^{\text{lp}}}^{\text{lp}})$ for which a linear underestimator needs to be computed. Assume further that variables that are currently fixed have been replaced by the corresponding constant.

Since $f(y)$ is concave, its convex envelope with respect to $[\underline{y}, \overline{y}]$ is vertex-polyhedral, that is, it is a polyhedral function which vertices correspond to the vertices of $[\underline{y}, \overline{y}]$. Therefore, any hyperplane $\alpha y + \beta$ that underestimates $f(y)$ in all vertices of $[\underline{y}, \overline{y}]$ is a valid linear underestimator. Maximizing $\alpha\hat{y} + \beta$ under these constraints gives an underestimator that is as tight as possible in the reference point $\hat{y}$. For the frequent cases $k = 1$ and $k = 2$, routines that directly compute such an underestimator are available. For $k > 2$, a linear program is solved.

Since such underestimators need to be computed repeatedly for varying domains, updates to the cut generating linear program (CGLP) are kept to a minimum. By use of the linear transformation $T : [0,1]^k \to [\underline{y}, \overline{y}]$ given by $T(\tilde{y})_i = \underline{y}_i + (\overline{y}_i - \underline{y}_i)\tilde{y}_i$, $i = 1, \ldots, k$, the matrix of the CGLP can be kept constant:

$$\begin{aligned} &\max_{\tilde{\alpha},\tilde{\beta}} \tilde{\alpha}^\top T^{-1}(\hat{y}) + \tilde{\beta}, \\ &\text{s.t. } \tilde{\alpha}^\top \tilde{y} + \tilde{\beta} \le f(T(\tilde{y})), \qquad \forall \tilde{y} \in \{0,1\}^k. \end{aligned}$$

The cut in the original y-space is then recovered via $\alpha_j = \frac{\tilde{\alpha}_j}{\overline{y}_j - \underline{y}_j}$, $j = 1, \ldots, k$, and $\beta = \tilde{\beta} - \sum_{j=1}^k \alpha_j \underline{y}_j$. Since the CGLP typically has more rows than columns, the dual of CGLP is formulated and solved. To increase the chance that $\alpha y + \beta$ is a facet of the convex envelope of $f(y)$, the reference point is perturbed and moved into the interior of $[\underline{y}, \overline{y}]$.

At the moment, underestimators for concave functions in more than 14 variables are not computed due to the size of the CGLP being exponential in k. In fact, the detection algorithm in the concave nonlinear handler already returns unsuccessful if the recognized concave expression has more than 14 variables. Dynamic row or column generation methods could be added to overcome this limit [11].

Since the underestimator may not be tight at $(\hat{y}, f(\hat{y}))$, all variables are registered as branching candidates by this nonlinear handler.

4.7 Nonlinear Handler for Quotients

Note that the available expression handlers (see Section 4.1) do not include a handler for quotients since they can equivalently be written using a product and a power expression. However, the default extended formulation for an expression $y_1 y_2^{-1}$ is given by replacing y_2^{-1} by a new auxiliary variable w. The linear outer-approximation is then obtained by estimating $y_1 w$ and y_2^{-1} separately. The quotient nonlinear handler can provide tighter estimates by checking whether a given function $h_i^{\text{lp}}(x)$ can be cast as

$$f(y) = \frac{ay_1 + b}{cy_2 + d} + e \tag{30}$$

with $a, b, c, d, e \in \mathbb{R}$, $a, c \neq 0$, and y_1 and y_2 being either original variables (x) or subexpressions of $h_i^{\text{lp}}(x)$. At the moment, only expressions $h_i^{\text{lp}}(\cdot)$ that are of the form $ay_1 y_2^{-1}$ or $ay_1 y_2^{-1} + by_2^{-1} + e$ are recognized as quotient and it is checked whether y_j equals $a'_j y'_j + b'_j$ for some a'_j, b'_j, $j = 1, 2$. For estimation and domain propagation, the univariate ($y_1 = y_2$) and bivariate ($y_1 \neq y_2$) cases are handled separately.

4.7.1 Univariate Quotients ($y_1 = y_2$)

If $-d/c \notin [\underline{y}_2, \overline{y}_2]$, then $f(y)$ is either convex or concave on $[\underline{y}, \overline{y}]$. Thus, under- and overestimators are computed via a tangent or a secant on the graph of $f(y)$. If the singularity is in the domain of y, then no estimator can be computed.

For forward domain propagation, observe that the minimum and maximum of $f(y)$ is attained at $\underline{y}$ or $\overline{y}$ if $-d/c \notin [\underline{y}_2, \overline{y}_2]$. It is therefore sufficient for evaluate $f(y)$ at $\underline{y}$ and $\overline{y}$ to obtain $f([\underline{y}, \overline{y}])$. If the singularity is in the domain of y, then no finite bounds on $f(y)$ can be computed.

For backward domain propagation, let $[\underline{f}, \overline{f}]$ be the bounds given for $f(y)$. Inverting (30) yields

$$y = \frac{b - d[\underline{f}, \overline{f}]}{c[\underline{f}, \overline{f}] - a}.$$

This interval can be evaluated as in forward propagation.

4.7.2 Bivariate Quotients

Let $\underline{x}, \overline{x}, \underline{y}, \overline{y}$ denote lower and upper bounds on variables x and y, respectively. In the bivariate case, estimators are computed for y_1'/y_2' and then transformed back into (y_1, y_2)-space. Thus, for the following, $a = 1$, $b = 0$, $c = 1$, $d = 0$ is assumed.

If $0 \in [\underline{y}_2, \overline{y}_2]$, then no estimators can be computed.

If $\underline{y}_1 \geq 0$ and $\underline{y}_2 > 0$, then an underestimator is obtained by computing a tangent on the graph of the convex underestimator of $f(y)$, which is given by Zamora and Grossmann [120] as

$$\frac{1}{y_2} \left(\frac{y_1 + \sqrt{\underline{y}_1 \overline{y}_1}}{\sqrt{\underline{y}_1} + \sqrt{\underline{y}_1}} \right)^2.$$

An overestimator is given by a hyperplane that passes either through the points $(\underline{y}_1, \underline{y}_2, \underline{y}_1/\underline{y}_2)$, $(\underline{y}_1, \overline{y}_2, \underline{y}_1/\overline{y}_2)$ and $(\overline{y}_1, \underline{y}_2, \overline{y}_1/\underline{y}_2)$ or through the points $(\underline{y}_1, \overline{y}_2, \underline{y}_1/\overline{y}_2)$, $(\overline{y}_1, \underline{y}_2, \overline{y}_1/\underline{y}_2)$ and $(\overline{x}_1, \overline{y}_2, \overline{y}_1/\overline{y}_2)$. The choice of the three points depends on which combination yields the estimator that is tightest at the given reference point, for example, the LP solution to be separated.

If $0 \in [\underline{y}_1, \overline{y}_1]$ and $\underline{y}_2 > 0$, recall that the nonlinear handler works on the constraint $y_1/y_2 \lesseqgtr_i w_i^{\text{lp}}$ in ($\text{MINLP}_{\text{ext}}^{\text{lp}}$) and that it is valid to replace $\lesseqgtr_i$ by $=$. Rewriting as $y_1 = y_2 w_i^{\text{lp}}$, linear McCormick envelopes [76] are computed for $y_2 w_i^{\text{lp}}$. These inequalities are then rearranged to obtain linear estimators on w_i^{lp}.

The cases of $\overline{y}_1 < 0$ and/or $\overline{y}_2 < 0$ are handled similarly, up to swapping and negation of bounds, coefficients, and inequality signs.

Since each variable appears only once in the expression, the default domain propagation for $f(y)$ does not suffer from the dependency problem of interval arithmetics. Therefore, the nonlinear handler does not provide an extra implementation of domain propagation for the bivariate case.

4.8 Nonlinear Handler for Perspective Reformulation

This nonlinear handler creates strengthened cutting planes for constraints that depend on semi-continuous variables. A variable x_j, $j \in \mathcal{N}$, is semi-continuous with respect to the binary indicator variable $x_{j'}$, $j' \in \mathcal{I}$, if it is restricted to the domain $[\underline{x}_j^1, \overline{x}_j^1]$ when $x_{j'} = 1$ and has a fixed value x_j^0 when $x_{j'} = 0$. In the rest of this subsection, the superscript 0 denotes the value of a semi-continuous variable at $x_{j'} = 0$.

Consider the constraint

$$h_i^{\text{lp}}(x, w_{i+1}^{\text{lp}}, \ldots) \lesseqgtr w_i^{\text{lp}} \tag{31}$$

and write $h_i^{\text{lp}}(\cdot)$ as a sum of its nonlinear and linear parts:

$$h_i^{\text{lp}}(x, w_{i+1}^{\text{lp}}, \ldots) = h_i^{\text{nl}}(x_{\text{nl}}, w_{\text{nl}}^{\text{lp}}) + h_i^{\text{l}}(x_{\text{l}}, w_{\text{l}}^{\text{lp}}),$$

where $h_i^{\text{nl}}(\cdot)$ is a nonlinear function, $h_i^{\text{l}}(\cdot)$ is a linear function, x_{nl} and $w_{\text{nl}}^{\text{lp}}$ are the vectors of variables x and w^{lp}, respectively, that appear only in the nonlinear part of h_i^{lp}, and x_{l} and w_{l}^{lp} are the vectors of variables x and w^{lp}, respectively, that appear only in the linear part of $h_i^{\text{lp}}(\cdot)$.

The perspective handler works on Constraint (31) if x_{nl} and $w_{\text{nl}}^{\text{lp}}$ are semi-continuous with respect to the same indicator variable $x_{j'}$, and at least one other nonlinear handler provides estimation (`ESTIMATE` callback) for $h_i^{\text{lp}}(\cdot)$. Thus, a nonlinear handler that implements only the `ENFO` callback, such as, for example, the SOC handler, is not suitable.

4.8.1 Detection of Semi-continuous Variables

To determine whether a variable x_j is semi-continuous, the detection callback searches for pairs of implied bounds on x_j with the same indicator $x_{j'}$:

$$x_j \leq \alpha^{(u)} x_{j'} + \beta^{(u)},$$
$$x_j \geq \alpha^{(\ell)} x_{j'} + \beta^{(\ell)}.$$

If $\beta^{(u)} = \beta^{(\ell)}$, then x_j is a semi-continuous variable and $x_j^0 = \beta^{(u)}$, $\underline{x}_j^1 = \alpha^{(\ell)} + \beta^{(\ell)}$, and $\overline{x}_j^1 = \alpha^{(u)} + \beta^{(u)}$.

This information can be obtained either from linear constraints in x_j and $x_{j'}$ or by finding implicit relations between x_j and $x_{j'}$. Such relations can be detected by probing, which fixes $x_{j'}$ to its possible values and propagates all constraints in the problem, thus detecting implications of $x_{j'} = 0$ and $x_{j'} = 1$. SCIP stores the implied bounds in a globally available data structure.

The perspective nonlinear handler also detects semi-continuous auxiliary variables. Given $h_i^{\text{lp}}(x, w_{i+1}^{\text{lp}}, \ldots) \lesseqgtr_i w_i^{\text{lp}}$, where $x, w_{i+1}^{\text{lp}}, \ldots$ are semi-continuous variables depending on the same indicator $x_{j'}$, the auxiliary variable w_i^{lp} can also be assumed to be semi-continuous since it is valid to replace $\lesseqgtr_i$ by $=$.

4.8.2 Separation

Suppose that the current relaxation solution violates Constraint (31). If the non-perspective nonlinear handlers claimed that estimators of $h_i^{\text{lp}}(\cdot)$ depend on variable bounds, probably because the functions is nonconvex, then probing is first performed for $x_{j'} = 1$ in order to tighten the implied bounds on variables $x, w_{i+1}^{\text{lp}}, \ldots$. Linear underestimators (for "$\leq$" constraints) or overestimators (for "$\geq$" constraints) that are valid when $x_{j'} = 1$ are then obtained for the tightened bounds. This estimator $\ell(\cdot)$ can be separated into parts corresponding to the nonlinear and linear variables of $h_i^{\text{lp}}(\cdot)$, respectively:

$$\ell(x, w_{i+1}^{\text{lp}}, \ldots) = \ell^{\text{nl}}(x_{\text{nl}}, w_{\text{nl}}^{\text{lp}}) + \ell^{\text{l}}(x_{\text{l}}, w_{\text{l}}^{\text{lp}})$$

An extension procedure is applied to the nonlinear part to ensure it is valid and tight for $x_{j'} = 0$, while the linear part can remain unchanged since it shares none of the variables with the nonlinear part:

$$\ell^{\text{nl}}(x_{\text{nl}}, w_{\text{nl}}^{\text{lp}}) + \left(h_i^{\text{nl}}(x_{\text{nl}}^0, w_{\text{nl}}^{\text{lp},0}) - \ell^{\text{nl}}(x_{\text{nl}}^0, w_{\text{nl}}^{\text{lp},0})\right)(1 - x_{j'}) + \ell^{\text{l}}(x_{\text{l}}, w_{\text{l}}^{\text{lp}}).$$

This extension ensures that the estimator is equal to $h_i^{\text{lp}}(x, w_{i+1}^{\text{lp}}, \ldots)$ for $x_{j'} = 0$, $x_{\text{nl}} = x_{\text{nl}}^0$, and $w_{\text{nl}} = w_{\text{nl}}^0$, and equal to $\ell(x, w_{i+1}^{\text{lp}}, \ldots)$ for $x_{j'} = 1$. In the convex case, cuts thus obtained are equivalent to the classic perspective cuts [28]. More details on the implementation in SCIP can be found in the paper by Bestuzheva et al. [17].

4.9 Separator for Cuts from the Reformulation-Linearization Technique

A nonlinearity that appears frequently is a product between two variables and/or functions. The separator for Reformulation-Linearization Technique (RLT) cuts [6, 7, 8] for bilinear product relations in ($\text{MINLP}_{\text{ext}}^{\text{lp}}$) and the separators discussed in the following two sections focus on enforcing the relationship between a product of two variables (original or auxiliary) and a corresponding auxiliary variable. The RLT separator can additionally reveal linearized products between binary and continuous variables.

There exist variations of the RLT that can be applied to any (not necessarily quadratic) polynomials [100]. This separator, however, deals with bilinear products only.

In the following, x refers to any variable of ($\text{MINLP}_{\text{ext}}^{\text{lp}}$) and $X_{i,j}$ refers to the auxiliary variable (w^{lp}) that is associated with a constraint $x_i x_j \lesseqgtr X_{i,j}$ in ($\text{MINLP}_{\text{ext}}^{\text{lp}}$). Note that $X_{i,j}$ may not exist in ($\text{MINLP}_{\text{ext}}^{\text{lp}}$) for every pair of x_i and x_j, even when $x_i x_j$ appears in some constraint of (MINLP) (for example, auxiliary variables are not created for terms in convex quadratic constraints). Both $X_{i,j}$ and $X_{j,i}$ refer to the same variable.

Given a product relation $X_{ij} = x_i x_j$, where $x_i \in [\underline{x}_i, \overline{x}_i]$, $x_j \in [\underline{x}_j, \overline{x}_j]$ and a linear constraint $a^\top x \leq b$, RLT cuts are derived by first multiplying the constraint by nonnegative bound factors $(x_i - \underline{x}_i)$, $(\overline{x}_i - x_i)$, $(x_j - \underline{x}_j)$, and $(\overline{x}_j - x_j)$. For instance, consider multiplication by the factor $(x_i - \underline{x}_i)$, which yields a valid nonlinear inequality:

$$a^\top x\,(x_i - \underline{x}_i) \leq b\,(x_i - \underline{x}_i). \tag{32}$$

This is referred to as the reformulation step.

The linearization step is then performed for all terms $x_k x_i$ in (32). If a product relation $X_{ki} = x_k x_i$ exists, then the product is replaced with X_{ki}. If x_k and x_i are contained in the same clique, the product is replaced with an equivalent linear expression. Otherwise, it is replaced by a linear under- or overestimator such that the inequality remains valid. By default, RLT cuts are constructed only for combinations of rows and bound factors where all relations $X_{ki} = x_k x_i$ exist (parameter `separating/rlt/maxunknownterms`).

4.9.1 Implicit Product Detection

Bilinear product relations in which one of the multipliers is binary can equivalently be written via mixed-integer linear constraints. Likewise, MILP constraints representing such relations can be identified in order to derive these implicit bilinear products.

Consider two linear constraints depending on the same three variables x_i, x_j and x_k, where x_i is binary:

$$a_1 x_i + b_1 x_k + c_1 x_j \leq d_1, \tag{33a}$$
$$a_2 x_i + b_2 x_k + c_2 x_j \leq d_2. \tag{33b}$$

If $b_1 b_2 > 0$ and $c_2 b_1 - b_2 c_1 \neq 0$, then these constraints imply a product relation,

$$A x_i + B x_k + C x_j + D \lesseqgtr x_i x_j, \tag{34}$$

where the coefficients A, B, C, and D and the inequality sign are obtained by:

- setting x_i to 1 in (33a) and (34), and requiring that the coefficients are similar for each variable, and the constants are equal;

– setting x_i to 0 in (33b) and (34), and similarly requiring equivalence;
– solving the linear system resulting from the first two steps.

SCIP analyses the linear constraints in the problem and stores all detected implicit products. RLT cuts that use these products may strengthen the default continuous relaxation $\{(x_i, x_j, x_k) : x_i \in [0,1], (33)\}$.

4.9.2 Separation

Let $(\hat{x}, \hat{X})$ be the solution to be separated. In order to reduce the computational cost of RLT cut separation, SCIP takes into account the signs of coefficients of linear constraints and signs of product relation violations. In particular, when multiplying a constraint $a^\top x \leq b$ by a bound factor, the resulting RLT cut can only be violated if $a_k \hat{x}_k \hat{x}_j < a_k \hat{X}_{kj}$, that is, when replacing the product with the corresponding variable increases the violation of the inequality. This fact is used to ignore combinations of linear constraints and bound factors that will not produce a violated cut, thus reducing the computational effort.

This is implemented via a row marking algorithm which, for every variable x_i that participates in bilinear products, iterates over all variables x_j that appear in products together with x_i. When it encounters a violated product, the algorithm iterates over all linear rows where x_j has a nonzero coefficient and stores them in a sparse sorted array together with the marks indicating which bound factors of x_i it should be multiplied with. The cut generation algorithm then iterates over the array of marked rows and constructs RLT cuts from the products of each row with the suitable bound factors.

More details on the algorithms and implementation will be included in the upcoming paper [**?**].

4.10 Separator for Principal Minors of $X \succeq xx^\top$

Another new separator that enforces bilinear product relations in ($\text{MINLP}^{\text{lp}}_{\text{ext}}$) is `sepa_minor`. The notation introduced in the previous section is used.

A convex relaxation of condition $X = xx^\top$ is given by requiring $X - xx^\top$ to be positive definite. Separation for the set $\{(x, X) : X - xx^\top \succeq 0\}$ itself is possible, but cuts are typically dense and may include variables X_{ij} for products that do not exist in the problem [86]. Therefore, `sepa_minor` considers only (principle) 2×2 minors of $X - xx^\top$, which also need to be positive semi-definite. By Schurs complement, this means that the condition

$$A_{ij}(x, X) \coloneqq \begin{bmatrix} 1 & x_i & x_j \\ x_i & X_{ii} & X_{ij} \\ x_j & X_{ij} & X_{jj} \end{bmatrix} \succeq 0 \tag{35}$$

needs to hold. The separator detects principle minors for which X_{ii}, X_{jj}, X_{ij} exist and enforces $A_{ij}(x, X) \succeq 0$.

To identify which entries of the matrix X exist, the separator iterates over the available nonlinear constraints. For each constraint, its expressions are explored and all expressions of the form x_i^2 and $x_i x_j$ are collected. Then, the separator iterates through the found bilinear terms $x_i x_j$ and if the corresponding expressions x_i^2 and x_j^2 exist, a minor is detected.

Let $(\hat{x}, \hat{X})$ be a solution that violates (35), i.e., there exists an eigenvector $v \in \mathbb{R}^3$ of $A_{ij}(\hat{x}, \hat{X})$ with $v^\top A_{ij}(\hat{x}, \hat{X})v < 0$. To separate $(\hat{x}, \hat{X})$, `sepa_minor` adds the globally valid linear inequality $v^\top A_{ij}(x, X)v \geq 0$ to the separation storage of SCIP.

For circle packing instances, the minor cuts are not really helpful [56]. Since experiments showed that SCIP's overall performance was negatively affected, circle packing constraints are identified and their bilinear terms are ignored by `sepa_minor` (parameter `separating/minor/ignorepackingconss`).

4.11 Separator for Intersection Cuts on Rank-1 Constraint for X

Another new separator that enforces bilinear product relations in ($\text{MINLP}^{\text{lp}}_{\text{ext}}$) is `sepa_interminor`. The notation introduced in Section 4.9 is used.

Since $X = xx^\top$ has rank 1 in any feasible solution, any 2×2 minor $\begin{pmatrix} X_{i_1 j_1} & X_{i_1 j_2} \\ X_{i_2 j_1} & X_{i_2 j_2} \end{pmatrix}$ of X needs to have determinant 0. That is, for any set of variable indices i_1, i_2, j_1, j_2 with $i_1 \neq i_2$ and $j_1 \neq j_2$, the condition

$$X_{i_1 j_1} X_{i_2 j_2} = X_{i_1 j_2} X_{i_2 j_1} \tag{36}$$

needs to hold. If all variables in this constraint exist in the problem and the solution $(\hat{x}, \hat{X})$ that is to be separated violates (36), the separation strategy described in Section 4.3.3 is used to add (strengthened) intersection cuts that separate $(\hat{x}, \hat{X})$. Additionally, it is also possible (parameter `separating/interminor/usebounds`) to use the bounds on x_{i_1}, x_{i_2}, x_{j_1}, x_{j_2} to improve the cut by enlarging the corresponding S-free set [21].

The separator is currently disabled by default.

4.12 Revised Primal Heuristic that Solves NLP Subproblem

The primal heuristic `subnlp` targets problems like (MINLP), but runs on any CIP where the NLP relaxation is enabled. Given a point $\tilde{x}$ that satisfies the integrality requirements ($\tilde{x}_i \in \mathbb{Z}$ for all $i \in \mathcal{I}$), the heuristic fixes all integer variables to the values given by $\tilde{x}$ in a copy of the CIP, presolves this copy, and triggers a solution of the NLP relaxation by an NLP solver using $\tilde{x}$ as starting point. If the NLP solver, such as Ipopt, finds a solution that is feasible (and often also locally optimal) for the NLP relaxation, it is tried whether it is also feasible for the CIP. If the CIP is a MINLP, then this should usually be the case. The starting point $\tilde{x}$ can be the current solution of the LP relaxation if integer-feasible, can be a point that a primal heuristic that searches for feasible solutions of the MILP relaxation has computed, or can have been passed on by other primal heuristics that look for MINLP solutions, such as `undercover` or `mpec`.

The `subnlp` primal heuristic, which is implemented in virtually any global MINLP solver, had been added to SCIP together with the support for quadratic constraints (SCIP 1.2.0). The rewrite of the algebraic expression system (Section 4.1) and the handling of nonlinear constraints (Section 4.2) and the updates to the NLP solver interfaces and NLP relaxation (Section 4.13) were a good opportunity for a thorough revision of the heuristic.

Starting Condition and Iteration Limit By default, the heuristic is called in every node of the branch-and-bound tree, but invoking an NLP solver whenever a starting point $\tilde{x}$ is available would be too costly. After the heuristic has been run, it therefore waits until a certain number of nodes have been processed. How many nodes these are depends on the success of the heuristic in previous calls, the number of iterations the NLP solver used in previous calls, and the iteration limit that would be imposed for the following NLP solve. Previously, the iteration limit was essentially static, which could mean that on problems with difficult NLPs a lot of effort was wasted on NLP solves that were interrupted by a too small iteration limit.

With SCIP 8, the heuristic tries to adapt the iteration limit to the NLPs to be solved. For that, the heuristic counts how often an NLP solve stopped due to an iteration limit (n^{iterlim}) and how often it finished successfully, that is, stopped because convergence criteria were fulfilled (n^{okay}). Let i^{iterlim} be the highest iteration limit used among all NLP solves that stopped due to an iteration limit and let i^{okay} be the total number of iterations used in all NLP solvers that finished successfully.

Further, let $i^{\min}$ be a minimal number of iterations that should be granted to every NLP solve (parameter `heuristics/subnlp/itermin` = 20). Finally, let n^{init} be the number of initial NLP solves that should be granted i^{init} many iterations (parameters `heuristics/subnlp/ninitsolves` = 2 and `heuristics/subnlp/iterinit` = 300). The iteration limit i^{next} for the next NLP solve is then decided as follows:

1. If $n^{\text{iterlim}} > n^{\text{okay}}$, then $i^{\text{next}} := \max(i^{\min}, 2i^{\text{iterlim}})$. That is, double the iteration limit if more solves ran into an iteration limit than were successful.
2. Otherwise, if $n^{\text{okay}} > n^{\text{init}}$, then $i^{\text{next}} := \max(i^{\min}, 2\frac{i^{\text{okay}}}{n^{\text{okay}}})$. That is, if there were a few successful solves to far, then use twice the average number of iterations spend in these solves as iteration limit.
3. Otherwise, if n^{okay}, then $i^{\text{next}} := \max(i^{\min}, i^{\text{init}}, 2\frac{i^{\text{okay}}}{n^{\text{okay}}})$. That is, consider also i^{init} if there had not been enough successful solves so far.
4. Otherwise, $i^{\text{next}} := \max(i^{\min}, i^{\text{init}})$.

To decide whether to execute the heuristic, an iteration contingent i^{cont} is calculated and checked against i^{next}. Compared to SCIP 7, this has received only minor updates:

1. Initialize $i^{\text{cont}} := 0.3$(number of nodes processed + 1600) (parameters `heuristics/subnlp/{nodesfactor,nodesoffset}`).
2. Weigh by previous success of heuristic: Let n^{tot} the total number of times the heuristic has run and n^{sol} the number of solutions found by the heuristic. If the heuristic ran a few times and is no longer in a phase where it tries to find a suitable iteration limit, then weigh i^{cont} by success of heuristics. That is, if $n^{\text{tot}} - n^{\text{iter}} > n^{\text{init}}$, then $i^{\text{cont}} := \frac{n^{\text{sol}}+1}{n^{\text{tot}}+1} i^{\text{cont}}$. Parameter $\beta :=$`heuristics/subnlp/successrateexp` allows to replace $\frac{n^{\text{sol}}+1}{n^{\text{tot}}+1}$ by $(\frac{n^{\text{sol}}+1}{n^{\text{tot}}+1})^{\beta}$.
3. Let i^{tot} be the total number of iterations used in all NLP solves (successful or not) so far. Then $i^{\text{cont}} := i^{\text{cont}} - i^{\text{tot}}$.
4. If $i^{\text{cont}} \geq i^{\text{next}}$, then the heuristic is run with i^{next} as iteration limit for the NLP solver.

Presolve The heuristic triggers a solve of the NLP relaxation of SCIP in a copy of the CIP. When the heuristic is run for a starting point $\tilde{x}$, integer variables are fixed to the values given in $\tilde{x}$, the current primal bound is set as cutoff, and SCIP's presolve is run with presolve emphasis set to "fast". The aim of the presolve is to propagate the fixing of the integer variables in the problem since many NLP solvers, in particular those that are interfaced by SCIP, only implement a very limited presolve. After presolve, if the problem is not empty or infeasible, SCIP is put into a state where its NLP relaxation can be solved. If the original CIP is a MINLP, then solutions that are feasible to this NLP relaxation should also be feasible in the original problem. Further, also solutions that are found during presolve are passed on to the original problem.

This process of fixing integer variables, setting a cutoff, and presolving the CIP repeats every time the heuristic is run. If, however, there are no binary or integer variables, then setting a cutoff and presolve is skipped and the copied problem is kept in a state where its NLP relaxation can be solved.

NLP Solve The NLP relaxation in the presolved copied CIP instance is solved by a NLP solver that is interfaced by SCIP. The solver is given $\tilde{x}$ as starting point and the iteration limit is set to i^{next}. If the NLP solver is Ipopt, then also the "expect infeasible problem" heuristic of Ipopt is enabled.

If the solver claims to have found a feasible solution, then it is tried to add this solution to the original problem. This can fail for three reasons: the objective function value is not good enough, the NLP relaxation is missing some constraints of the original

CIP, or the solution is only slightly infeasible due to presolve reductions. For example, due to tolerances, bounds of aggregated variables might be slightly violated. To work around this case, if a solution is not accepted, its objective value is not worse than the current primal bound, its maximal constraint violation is close to the feasibility tolerance, and the copied problem has been presolved ($\mathcal{I} \neq \emptyset$), then the NLP is resolved with a tightened feasibility tolerance (parameter `heuristics/subnlp/feastolfactor`). For this resolve, warmstart from the previous solution is enabled and the iteration count of the previous NLP solve is used as iteration limit. If the NLP resolve succeeds and produces a solution that is accepted in the original problem, then the tightened feasibility tolerance is used for all following NLP solves by the heuristic.

4.13 NLP Relaxation and Interfaces to NLP Solvers and Automatic Differentiation

The updated expressions framework (Section 4.1) triggered a revision of the NLP relaxation and the interfaces to NLP solvers (`NLPI`) and automatic differentiation (`EXPRINT`).

4.13.1 NLP Relaxation

The rows of the NLP relaxation (`SCIP_NLROW`) no longer distinguish a quadratic part. Therefore, rows now have the form

$$\text{left-hand side} \leq \text{linear terms} + \text{nonlinear term} \leq \text{right-hand side}$$

where the nonlinear term is given as a SCIP expression. When the nonlinear constraint handler (see Section 4.2) creates an NLP row for a constraint $\underline{g} \leq g(x) \leq \overline{g}$ of (MINLP), it separates linear terms from $g(x)$. The constraint handlers `and`, `bounddisjunction`, `knapsack`, `linear`, `linking`, `logicor`, `setppc`, and `varbound` now add themselves to the NLP relaxation. Previously, `and`, `bounddisjunction`, and `linking` constraints were not added. For `bounddisjunction`, only univariate constraints are added.

Further, it is pointed out that the NLP relaxation of SCIP is no longer based on the extended formulation ($\text{MINLP}^{\text{lp}}_{\text{ext}}$), but is now closer to the continuous relaxation of the original problem (MINLP).

4.13.2 Interfaces to NLP Solvers

Since expression handlers are now proper SCIP plugins that require a SCIP pointer for many operations and since expressions are used to specify NLPs, also the NLP solver interfaces (`NLPI`) are now proper SCIP plugins that require a SCIP pointer. However, as before, the NLPs that are specified via an NLPI can be independent of the problem that is solved by SCIP. For the expressions in the objective and constraints of such an NLP this means that the "var" expression handler, which refers to a SCIP variable (`SCIP_VAR*`), cannot be used. Instead, the handler for "varidx" expressions, which refer to a variable index, needs to be used. As a consequence, the evaluation and differentiation methods of expressions, which work with a SCIP solution (`SCIP_SOL`), are not available (the `EVAL` callback of the "varidx" expression handler raises an error). Instead, the NLP solver interfaces either implement their own evaluation and differentiation or resort to the helper functions implemented in `nlpioracle.{h,c}`.

Next to the adjustments to the new expressions framework, further updates and removals to the NLPI callbacks were implemented. For a detailed list, see the `CHANGELOG`. A notable change, though, is that parameter settings that specify the working limits and tolerances of an NLP solve are now passed directly to the `NLPISOLVE` callback and,

thus, are used for the corresponding solve only. The same applies to the NLP relaxation of SCIP and `SCIPsolveNLP()` (now a macro). The default values for the NLP solve parameters are now uniform among all NLP solvers and some parameters were added, removed, or renamed. The solve statistics now include information on the violation of constraints and variable bounds of the solution, if available.

The problem and optimization statistics that SCIP collects and prints on request (`display statistics`) now include a table for each used NLP solver, which prints the number of times the solver was used, the time spend, and how often each termination and solution status occurred. Additionally, the time spend for evaluation and differentiation can be shown (parameter `timing/nlpieval`).

As before, SCIP includes interfaces to the NLP solvers FilterSQP, Ipopt, and WORHP. In particular the interface to Ipopt has been improved. Only some points are mentioned here:

- Warmstarts from a primal/dual solution pair, either set via `NLPISETINITIALGUESS` or by using the solution from the previous solve, are now available. Further, Ipopt is instructed to reinitialize less datastructures if the structure of the NLP did not change since the last solve.
- When Ipopt requests an evaluation of the Jacobian or Hessian, function reevaluation is now skipped if possible.
- When Ipopt stops at a point that it claims to be locally infeasible, it is now checked whether the solution proves infeasibility, see Berthold and Witzig [15, Theorem 1]. If that is not the case, the solution status is changed to "unknown".
- A few Ipopt parameters can now be set directly via SCIP parameters (`nlpi/ipopt/*`).
- Due to changes in how the Ipopt output is redirected into the SCIP log, the Ipopt banner was no longer printed reliably for the first run of Ipopt anymore. Therefore, the banner has now been disabled completely.

4.13.3 Interface to Algorithmic Differentiation

For the computation of first and second derivatives, SCIP traditionally relied on a third-party automatic differentiation (AD) library. With the new expressions framework (Section 4.1), first derivatives and Hessian-vector products are available in SCIP itself. Their implementation relies on the `BWDIFF`, `FWDIFF`, and `BWFWDIFF` callbacks of the expression handlers. The latter two are not implemented for every expression handler so far. However, some NLP solvers make use of full Hessians and their sparsity pattern, something that is not available in the expressions framework itself yet. Further, the current datastructure for expressions with its many pointer-redirections does not perform too well when a fixed expression needs to be evaluated repeatedly in many points. Therefore, a separate AD library is still used in the interfaces to NLP solvers.

Currently, the only library that is interfaced is CppAD[5]. In the CppAD interface, a given expression is compiled into the serial datastructure (the "tape") that is used by CppAD. Here, expression types (i.e., which handler is used) are checked and translated into a form that is native to CppAD when possible. Since the CppAD interface is used by NLPIs only, it only supports the "varidx" expression and not the "var" expression (see begin of previous section). With SCIP 8, CppAD's feature to optimize the tape has been enabled.

Mapping of expression handlers to CppAD's operator types is available for all expression handler that are included in SCIP. For some expression types, such as signpower, this translation has been improved to avoid repeated recompilation of an expression. For expression handlers that are not known to the CppAD interface, the

[5] `https://github.com/coin-or/CppAD`

backward- and forward-differentiation callbacks of the expression handler are used to provide first derivatives. However, second derivatives (Hessians) are not yet available. In the IPOPT interface, the Hessian approximation will be activated in this case.

With SCIP 7, quadratic functions, including their derivatives, were treated differently from other nonlinear function. Further, the NLPs to be solved were build from the extended, thus sparse, formulation ($\text{MINLP}^{\text{lp}}_{\text{ext}}$). Therefore, nonlinear functions typically depended on only a few variables and, thus, it was usually sufficient to work with dense Hessians. With SCIP 8, though, also the derivatives of quadratics are computed by the AD library and the NLPs to be solved are closer to the original form (MINLP). For these reasons, CPPAD's routines to compute sparse Hessians are used now unless more than half of the Hessian entries are nonzero.

4.14 Performance Impact of Updates for Nonlinear Constraints

While Section 2.3 compared the performance of SCIP 7.0 and SCIP 8.0 on a set of MINLP instances, this section takes a closer look on the effect of replacing only the handling of nonlinear constraints in SCIP. That is, here the following two versions of SCIP are compared:

classic: the main development branch of SCIP as of 23th of August 2021; in this version, nonlinear constraints are handled as it has been in SCIP 7.0, with just a few bugfixes added;

new: as classic, but with the handling of nonlinear constraints replaced as detailed in this section and symmetry detection extended to handle nonlinear constraints (see Section 3.2.2).

For this comparison, SCIP has been build with GCC 7.5.0 and uses PAPILO 1.0.2 for MILP presolves, BLISS 0.73 to find graph automorphisms, CPLEX 20.1.0.1 as LP solver, IPOPT 3.14.4 as NLP solver, CPPAD 20180000.0 for automatic differentiation, and Intel MKL 2020.4.304 for linear algebra (LAPACK). IPOPT uses the same LAPACK and HSL MA27 as linear solver. All runs are carried out on identical machines with Intel Xeon CPUs E5-2660 v3 @ 2.60GHz and 128GB RAM in a single-threaded mode. As working limits, a time limit of one hour, a memory limit of 100000MB, an absolute gap tolerance of 10^{-6}, and a relative gap tolerance of 10^{-4} are set. All 1678 instances of MINLPLib (version 66559cbc from 2021-03-11) that can be handled by both the classic and the new version are used. It is noted that MINLPLib is not designed to be benchmark set, though, since, for example, some models are overrepresented with a large number of instance. For each instance, two additional runs where the order of variables and constraints were permuted by SCIP were conducted. Thus, in total 5034 jobs were run for each version of SCIP.

Table 4 summarizes the results. A run is considered as failed if the reported primal or dual bound conflicts with best known bounds for the instance, the solver aborted prematurely due to a fatal error (for example, failure in solving the LP relaxation of a node), or the solver did not terminate at the time limit. For this comparison, runs where the final solution is not feasible are accounted separately. One can observe that with the new version, for much fewer instances the final incumbent is not feasible for the original problem, that is, the issue discussed in Section 4.2.1 has been resolved for nonlinear constraints. For the remaining 49 instances, typically small violations of linear constraints or variable bounds occur. Further, the reduction in "failed" instances by half shows that the new version is also more robust regarding the computation of correct primal and dual bounds. Finally, we see that the new version solves about 400 additional instances than the classic one, but also does no longer solve about 200 instances within the time limit.

Subset	instances	metric	classic	new	both
all	5034	solution infeasible	481	49	20
		failed	143	70	18
		solved	2929	3131	2742
		time limit	1962	1833	1598
		memory limit	0	0	0
clean	4839	fastest	3733	3637	2531
		mean time	75.9s	70.3s	
		mean nodes	2543	2601	
$[0, 3600)$	2742	fastest	1990	1697	945
		mean time	4.7s	5.4s	
		mean nodes	415	455	
$[10, 3600)$	985	fastest	618	554	187
		mean time	55.6s	66.0s	
		mean nodes	3960	4502	
$[100, 3600)$	484	fastest	292	262	70
		mean time	185.3s	231.9s	
		mean nodes	12620	17150	
$[1000, 3600)$	141	fastest	72	81	12
		mean time	803.5s	623.5s	
		mean nodes	43345	39014	

Table 4: Comparison of performance of SCIP with classic versus new handling of nonlinear constraints on MINLPLib.

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

For the remaining subsets, $[t_1, t_2)$ refers to all instances where at least one version ran for t_1 or more seconds and both versions terminated in less than t_2 seconds. That is, only instances that could be solved to optimality by both versions are considered. For most of these subsets, the new version is still slower more often and on average than the classic version. Further, for a third of the instances that can be solved, both versions perform similar. Only on the (rather small) subset $[1000, 3600)$ of difficult-but-solvable instances does the new version improve.

Figure 4 shows performance profiles that compare both versions w.r.t. the time to solve an instance and the gap at termination. The time comparison visualizes what has already been observed in Table 4: the new version solves more instances, but can be slower. The gap comparison shows that on instances that are not solved, often the new version gives a smaller optimality gap than the classic version.

Appendix A provides detailed results on the performance of both SCIP versions on the considered MINLPLib instances. Further, information on the usage of the nonlinear handlers and separators that were described in this section is given.

5 SoPlex

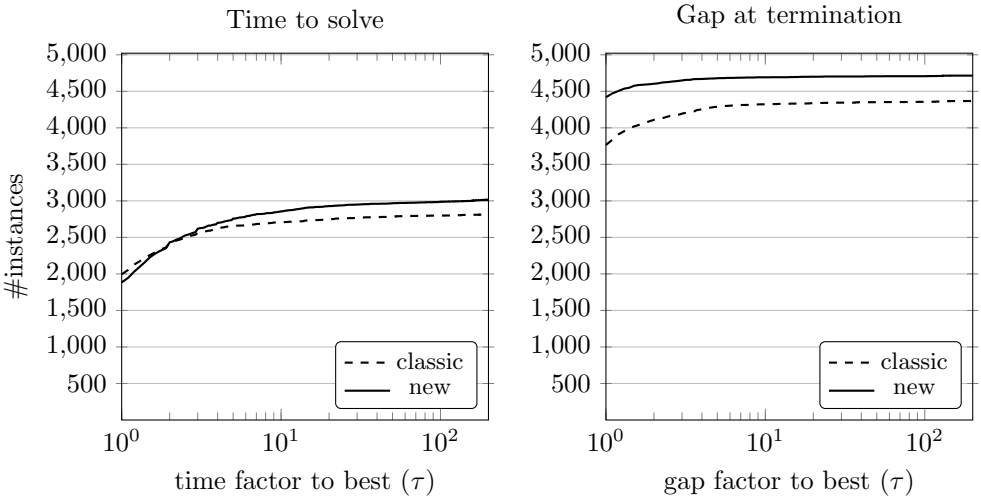

Figure 4: Performance profiles on time and gap at termination of SCIP with classic versus new handling of nonlinear constraints on MINLPLib. For a specific τ, the ordinate shows the number of instance for which the corresponding version of the solver was at most this much worse (regarding time (left) or gap at termination (right)) as the best of both versions. For the time plot, all instances that were solved to optimality are considered for each version. For the gap plot, all instances that did not fail are considered for each version.

5.1 Integration of PaPILO in SoPlex

As described in Section 6.1, version 2.0.0 of PaPILO supports postsolving of dual LP solutions and basis information. This makes it possible to integrate PaPILO fully as a presolving library into SoPlex. In version 6.0 of SoPlex, PaPILO is available as an additional option for presolving. The previous presolving implementation continues to be the default.

The PaPILO integration is handled similarly as in SCIP [36]. SoPlex calls a newly added simplifier plugin that converts the current problem to PaPILO's data structure and then calls the presolve routine. The changes from PaPILO's reduced problem are communicated by deleting the current matrix in SoPlex and subsequently recreating it from PaPILO's reduced problem, if the number of columns or rows decreased.

5.2 Technical Improvements

Several other smaller changes and improvements have been made in SoPlex 6.0. First, SoPlex was extended by a C interface, as explained in Section 7. Second, a rework of the internal data structures was necessary to fix warnings that were issued by current compiler versions. Third, the dependency on the Boost program options library has been removed and the command line interface has been restored to its classic version.

Furthermore, it is now possible to use SoPlex rational solving mode without linking a GMP library, using Boosts internal implementation of rational numbers. Finally, an ongoing LP solve of SoPlex can now be interrupted from a different thread, by calling the `setInterrupt` function of SoPlex. On the SCIP side, this is handled by calling `SCIPinterruptLP`.

6 PaPILO

PaPILO, a C++ library, provides presolving routines for MILP and LP problems and was introduced with SCIP Optimization Suite 7.0 [36]. PaPILO's transaction-based design generally allows presolvers to run in parallel without requiring expensive copies of the problem and without special synchronization in the presolvers themselves. Instead of applying the results immediately, presolvers return their reductions to the core, where they are applied in a deterministic, sequential order. Modifications in the data structure are tracked to avoid applying conflicting reductions. These conflicting reductions are discarded.

The main new feature in PaPILO 2.0 is support for postsolving dual and basis information, which is described in Section 6.1. This feature allows to use PaPILO as an integrated presolving library in SoPlex, see Section 5.1. Furthermore, PaPILO 2.0 comes with several improvements to the existing code base and presolving routines, described in Section 6.2. These changes result in a five percent improvement in the runtime when compared to the previous release.

6.1 Postsolving Dual LP Solutions and Basis Information

After removing, substituting, and aggregating variables from the original problem during presolving, the reduced problem (and solution) does not contain any information on missing variables. To restore the solution values of these variables and obtain a feasible original solution, corresponding data needs to be stored during the presolving process. The process of recalculating the original solution from the reduced one is called postsolving or post-processing [5].

Until version 1.0.2, PaPILO supported only postsolving primal solutions. In the latest version, PaPILO supports postsolving also for the dual solutions, reduced costs, the slack variables of the constraints, and the basic status of the variables and constraints for the presolvers: `DominatedColumns`, `Dualfix`, `ParallelCols`, `ParallelRows`, `Propagation`, `FixContinuous`, `ColSingleton`, and `SingletonStuffing`. These form the majority of the LP presolvers. The remaining presolvers are either only active in the presence of integer variables[6] or need to be disabled by the user[7].

Furthermore, in dual postsolve mode PaPILO only applies variable bound tightenings when they fix a variable. Otherwise, the solution to the reduced problem may correspond to a non-vertex solution in the original space and simple postsolving without an expensive crossover may not be possible. If the basic information is irrelevant for the user, the variable tightening without fixing can be turned on by setting the parameter `calculate_basis_for_dual` to false. An exception here is if a variable is unbounded. In this case, the bound of this variable is set to a finite value, which is slightly worse than the best possible bound so that the bound can not be tight in the reduced problem. This applies only to instances with no integer variables. Variable tightening is still performed for mixed-integer programs.

For primal postsolving only information about removed, substituted and aggregated variables needs to be tracked. By contrast, dual postsolving needs to be informed about every modification found during presolving. PaPILO 2.0 keeps tracks of these changes and saves them in the postsolve stack analogously to primal postsolving. For example, a row-bound change can lead to changes in the dual solution due to complementary slackness.

[6] Presolvers only active for MILP: `CoefficientStrenghtening`, `ImpliedInt`, `Probing`,`SimpleProbing`, `SimplifyInequalities`

[7] LP-Presolvers not supporting dual postsolving: `DualInfer`, `SimpleSubstitution`, `Substitution`, `Sparsify`, `ComponentDetection`, `LinearDependency`, see also settings file `lp_presolvers_with_basis.set` in the PaPILO repository

After postsolving, PaPILO checks if the original solution passes the primal and dual feasibility checks and fulfills the Karush-Kuhn-Tucker conditions [59] for LP. The result of the checks is logged to the console. Since also infeasible solutions can be postsolved, PaPILO does not abort if the checks fail and instead returns the result to the calling method.

For debugging purposes, this check can be performed after every step in the postsolve process. To activate this debugging feature, PaPILO needs to be built in debug mode and the parameter `validation_after_every_postsolving_step` has to be turned on. This may be expensive because the problem at the current stage needs to be calculated from the original problem by applying all reductions until this point.

The introduction of dual postsolving allows using PaPILO as presolving library in SoPlex. Section 5.1 contains a brief description of the integration.

6.2 Further Improvements

In this section we describe several smaller improvements in PaPILO 2.0. These changes affect mostly only the performance of PaPILO and rarely change the resulting reduced problem. All in all, these changes improve performance of PaPILO since the last release by about five percent performance (using 16 threads) in terms of runtime and number of presolving rounds, see Table 5 for details.

- When PaPILO 1.0 is run with only one thread, the presolvers are executed in sequential order, but the reductions of every presolver are only applied at the end of the presolving round. This is part of the parallel design of PaPILO and helps to guarantee deterministic results independently of the number of threads used. However, in sequential mode this does not guarantee best performance.

  Instead, when PaPILO 2.0 is run with only one thread, the reductions are applied before the next presolver, so that the next presolver can work on the modified problem. This feature can be turned off by setting the parameter `presolve.apply_results_immediately_if_run_sequentially` to false.

- `DualFix` handles an additional case with two conditions: first, the objective value of the variable is zero; second, if the variable has only up-/down-locks, the lower-/upperbound is (negative) infinity. Then, the variable can be set to infinity and deleted from the model. PaPILO removes the variable and marks all constraints containing the variable as redundant. In postsolving, the variable is set to the maximum/minimum value such that the variable bounds and the constraints in which it appeared in the original problem are not violated and hence, the solution stays feasible.

- PaPILO uses a transaction-based design to allow parallelization within the presolvers. This may generate conflicts when applying the reductions of the presolvers to the core. Conflicting reductions need to be discarded since it can not be ensured that the reduction is still valid. Conflicts make additional runs necessary to check if the discarded or a reformulated reduction can still be applied. Therefore, we performed a detailed analysis of the most prominent conflict relationships, introduced a new reduction type in PaPILO 2.0, and rearranged the order of presolving reductions reductions. In more detail, the improvements are as follows:

  · `ParallelRowDetection` could generate unnecessary conflicts mainly for `ParallelColDetection`. To avoid these reductions and additional runs, two new reduction types `RHS_LESS_RESTRICTIVE` and `LHS_LESS_RESTRICTIVE` were introduced. In contrast to `RHS` and `LHS`, the columns of a row are not marked as modified, if the initial bound was (negative) infinity.

Table 5: Performance comparison for PaPILO on MIPLIB 2017 benchmark

		PaPILO 1.0.3		PaPILO 2.0.0			
subset	instances	time [s]	rounds	time [s]	relative	rounds	relative
[0,tilim]	240	0.294557	19.57	0.281663	0.956	18.50	0.945
[0.01,tilim]	206	0.349799	22.15	0.334085	0.955	20.91	0.944
[0.1,tilim]	111	0.693586	33.02	0.660184	0.952	30.88	0.935
[1,tilim]	26	2.633462	43.69	2.477353	0.941	40.73	0.932

on a Intel Xeon CPU E5-2690 v4 @ 2.60GHz, 128GB - using 16 threads

  - `ParallelRowDetection`, `ParallelColDetection` and, `DominatedCol` could generate internal conflicts, if multiple rows/columns were parallel or dominating each other. To avoid these conflicts, bunches of parallel and dominating columns/rows are handled separately.
  - The order in which the reductions are applied to the core impacts the number of conflicts between the presolvers. We analyzed the conflicts between the presolvers and implement a new default order that minimizes the conflicts between the presolvers.

  The positive impact of these changes can be observed in the reduced number of rounds reported in Table 5.
- `SimpleSubstitution` handles an additional case to detect infeasibility faster.
- The loops in which the presolvers `ConstraintPropagation`, `DualFix`, `SimplifyInequality`, `CoefficientStrengthening`, `SimpleSubstitution`, `SimpleProbing`, and `ImpliedInteger` scan the rows or columns of the problems were parallelized. Hence, these presolvers can distribute their workload on different threads and exploit multiple threads internally.

Finally, two further features were introduced, improving transparency as well as the ability to debug:

- For analysis and debug purposes, PaPILO can now log every transaction in the order they were applied to the problem if verbosity level `kDetailed` is specified.
- PaPILO provides an additional way to validate its correctness. A feasible debug solution can be passed via the command-line parameter `-b`. After presolving the corresponding instance, PaPILO checks if the debug solution is still contained in the reduced problem and if the reduced solution can be postsolved to the same solution passed via command-line. It is recommended to turn off presolvers that use duality reasoning to (correctly) cut off optimal solutions.

7 Interfaces

SCIP is available via interfaces to several programming languages. These interfaces allow users to programmatically call SCIP with an API close to the C one or leverage a higher-level syntax. The following interfaces are available:

- The Python interface PySCIPOpt, which can now also be installed as a Conda package;
- The AMPL interface that comes as part of the main SCIP library and executable;
- The Julia package SCIP.jl;
- C wrapper for SoPlex;
- A Matlab interface.

We highlight below the main changes and development on interfaces to SCIP.

7.1 AMPL

The AMPL interface of SCIP has been rewritten and moved from being a separate project (`interfaces/ampl`) to being a part of the main SCIP library and executable (`src/scip`). The interface consists of a reader for `.nl` files as they are generated by AMPL and a specific AMPL-mode for the SCIP executable.

The `.nl` reader now relies on ampl/mp[8] instead of the AMPL solver library (ASL) to read `.nl` files. Required source files of ampl/mp are redistributed with SCIP. Therefore, building the `.nl` reader and the AMPL interface is enabled by default. The `.nl` reader supports linear and nonlinear objective functions and constraints, continuous, binary, and integer variables, and special-ordered sets. More than one objective function is not supported by the interface. A nonlinear objective function is reformulated into a constraint. In nonlinear functions, next to addition, subtraction, multiplication, and division, operators for power, logarithm, exponentiation, sine, cosine, and absolute value are supported. Variable and constraint flags (initial, separate, propagate, and others) can be set via AMPL suffixes.

If the SCIP executable is called with `-AMPL` as second argument, it expects the name of a `.nl` file (with `.nl` extension excluded) as first argument. In this mode, a SCIP instance is created, a settings file `scip.set` is read, if present, the `.nl` file is read, the problem is solved, an AMPL solution file (`.sol`) is written, and SCIP exists. Two additional parameters are available in the AMPL mode: boolean parameter `display/statistics` allows to enable printing the SCIP statistics after the solve; string parameter `display/logfile` allows to specify the name of file to write the SCIP log to. If the problem is an LP, SCIP presolve has not run, and the LP was solved, then a dual solution is written to the solution file, too.

7.2 Julia

The Julia package `SCIP.jl` has been in development since SCIP 3 with several improvements since SCIP 7. It contains a lower-level interface matching the SCIP public C API and a higher-level interface based on `MathOptInterface.jl` (MOI)[60]. The lower-level interface is automatically generated with `Clang.jl` to match the public SCIP C API, allowing for the direct conversion of C programs using SCIP into Julia ones. `MathOptInterface.jl` is a uniform interface for constrained structured optimization in Julia. Solvers specify the types of constraints they support and implement those only. Users can use the common interface for multiple solvers across different classes of problems including LP, MILP, (mixed-integer) conic optimization problems. Higher-level modeling languages such as `JuMP.jl` are implemented on top of MOI, allowing practitioners to define their optimization model in a syntax close to the mathematical specification and solve it through SCIP or swap solvers in a single line.

The `SCIP.jl` package can also automatically download the appropriate compiled binaries for SCIP and some of its dependencies on some platforms. This removes the need for users to download and compile SCIP separately. Custom SCIP binaries can still be passed to the Julia package when building it. This integration was made possible by cross-compiling SCIP through the `BinaryBuilder.jl` infrastructure, creating binaries for multiple combinations of OS, architecture, C runtime, and compiler. The binaries are available through GitHub and versioned for other platforms to use outside of Julia.

[8] `https://github.com/ampl/mp`

7.3 C Wrapper for SoPlex

With SCIP 8, there also comes a C wrapper for SoPlex. Since in some environments it is much easier to interface with C code than it is with C++ (which is the language SoPlex is written in), this wrapper paves the way for other projects to use SoPlex as a standalone LP solver and not only through SCIP. By building a pure C, simple shared library and header file, it is now possible to easily call SoPlex through the foreign function interface from many other languages.

7.4 Matlab

In the past, two interfaces from Matlab to SCIP existed. SCIP came with a rudimentary Matlab interface and there was the `OPTI Toolbox` by Jonathan Currie, available at `https://github.com/jonathancurrie/OPTI`. However, the development of the OPTI Toolbox stopped. In order to retain the advantages of this interface, a new interface was based on it. This new interface is available through the Git repository

`https://github.com/scipopt/MatlabSCIPInterface`.

The following changes have been implemented:

- The new interface also runs under Linux and MacOS.
- It works for Octave (but note that at the time of writing there is a bug in Octave that blocks the usage of the nonlinear part).
- The support of other solvers has been stripped away, but the interface to SCIP has been revised and adapted to SCIP 8.0.
- The interface now also fully works for SCIP-SDP.

For the installation details, we refer to the corresponding web page. However, the installation should only require calling

`matlabSCIPInterface_install.m` or `matlabSCIPSDPInterface_install.m`

from Matlab or Octave. Then the interface will be build and you are possibly asked where to find the SCIP or SCIP-SDP installation (you can also supply this information through the environment variables `SCIPDIR`/`SCIPOPTDIR` or `SCIPSDPDIR`).

To highlight the advantages of this interface, we briefly show an example. To solve the NLP

$$\min_{x \in \mathbb{R}^2} \{(x_1 - 1)^2 + (x_2 - 1)^2 : 0 \leq x_1, x_2 \leq 2\},$$

one can use the symbolic definition of the objective function:

```
obj = @(x) (x(1) - 1)^2 + (x(2) - 1)^2;
```

The remaining code is

```
lb = [0.0; 2.0];
ub = [0.0; 2.0];
x0 = [0.0, 0.0];
Opt = opti('obj',obj,'lb',lb,'ub',ub)
[x,fval,exitflag,info] = solve(Opt,x0)
```

More examples are available in the repository.

8 ZIMPL

ZIMPL 3.5.0 is released with the release of SCIP Optimization Suite 8.0. ZIMPL now allows also nonlinear objective functions. There has been quite some work to increase the code quality further by augmenting the code with compiler attributes. Also the work has been started to completely switch to C99 declarations, i.e, move the variable declarations from the start of the functions further inside, use local loop variables, const as much as possible, and in general try to initialize variables once they are created. Getting the maximum out of gcc, clang, clang-analyzer, and pclint is an interesting experiment, which however clearly shows that C was never mend to be validated.

A major additional feature in ZIMPL is the ability to write out suitable instances as a Quadratic Unconstrained Binary Optimization (QUBO) problem. Unfortunately, there is no standard format for QUBO files yet, so we support several small varieties of a sparse format for the moment. Furthermore, we have started to implement the ability to automatically convert constraints into quadratic binary objective functions [43]. With this release it is just a first experimental and limited ability, but we plan to extend this continuously with the next releases.

9 The UG Framework

UG is a generic framework for parallelizing branch-and-bound based solvers in a distributed or shared memory computing environment. It was designed to parallelize powerful state-of-the-art branch-and-bound based solvers (we call these "*base solvers*". Originally, the base solver is a branch-and-bound based solver, but in this release, it is redefined as any solver that is being parallelized by UG) externally in order to exploit their powerful performance. UG has been developed over 10 years as beta versions to have general interfaces for the base solvers. Internally, we have developed parallel solvers for SCIP [102, 106, 103], CPLEX (not developed anymore), FICO Xpress [104], PIPS-SBB [79, 80], Concorde[9], and QapNB [29]. In addition to the parallelization of these branch-and-bound base solvers, UG was used to develop MAP-SVP [108], which is a solver for the Shortest Vector Problem (SVP), and whose algorithm does not rely on branch-and-bound. Developers of several solvers parallelized by UG needed to internally modify the UG framework itself, since UG could not handle the base solvers directly. Especially, a success of MAP-SVP, which updated several records of the SVP challenge[10], motivated us to develop *generalized UG*, in which all solvers developed so far can be handled by a single unified framework. The generalized UG is included in this version of SCIP Optimization Suite as UG version 1.0.

UG version 1.0 is completely different from the previous versions internally, though its interfaces for branch-and-bound base solvers remain the same as far as possible. Figure 5 shows the class hierarchy of UG version 1.0. The original UG base classes are separated into branch-and-bound related codes and the others, so that non-branch-and-bound solvers can be parallelized naturally. In the original UG, the `ParaSolver` class, which wraps the "base solver", and the `ParaComm` class, which warps communication codes or parallelization libraries, are abstracted. On top of these abstractions, in UG version 1.0, the `ParaLoadCoordinator` class, which is a controller of the parallel solver, and the `ParaParamSet` class, which defines the parameter set, are also abstracted so that a "base solver" specific parallel algorithm can be implemented flexibly with the "base solver" specific parameters. The flexibility of UG version 1.0 can be observed in the paper of CMAP-LAP (Configurable Massively Parallel solver framework for LAttice Problems) [109], which is another parallel solver framework for lattice problems. On top

[9] `https://www.math.uwaterloo.ca/tsp/concorde.html`
[10] `http://latticechallenge.org/svp-challenge`

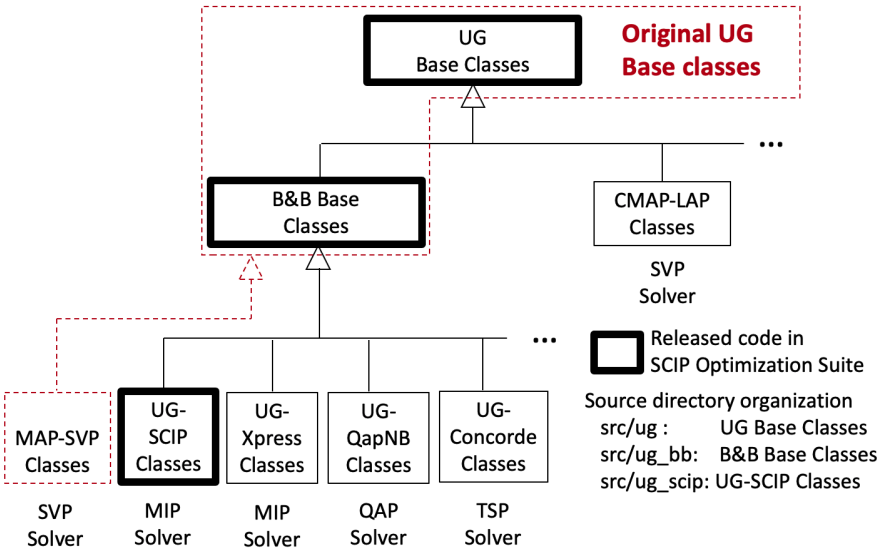

Figure 5: Class hierarchy and source code directory organization of the UG version 1.0

of CMAP-LAP, CMAP-DeepBKZ [**?**] has been developed, which is the successor of MAP-SVP and the first application of the generalized UG.

For the UG version 1.0, proper documentation of software is started, and Doxygen style documentation is introduced. Moreover, a CMake build system is included. In this opportunity, we made the following several modifications in FiberSCIP architecture and added a selfsplit ramp-up feature to FiberSCIP and ParaSCIP.

9.1 Join `ParaSolver` Threads of FiberSCIP

The *ramp-up* is a process which runs until all CPU cores become busy. For a general discussion of the ramp-up process for parallel branch-and-bound, see Ralphs et al. [87] and for the ramp-up process of FiberSCIP see Shinano et al. [106]. One of the distinguishing features of FiberSCIP is racing ramp-up. FiberSCIP is composed of a `ParaLoadCoordinator` thread and several `ParaSolver` threads. During the ramp-up phase, all ParaSolver threads solve the same root node with different parameter settings until certain termination criteria is met, that is, FiberSCIP generates multiple search trees in parallel and selects the winner within the `ParaSolver` threads, and afterward the winner search tree is solved in parallel. FiberSCIP is composed of an LC thread and several ParaSolver threads.

In previous versions, the `ParaSolver` threads were detached. The reason why the `ParaSolver` threads were detached is to enable terminating FiberSICP as soon as possible when one of the racing `ParaSolver` threads has solved the instance. When we developed FiberSCIP for the first time, it was very hard to interrupt SCIP when an LP solve is being executed. Therefore, all `ParaSolver` threads were detached, and the main thread exits when one of the `ParaSolver` threads has solved the instance. However, this mechanism leads to some instability in FiberSCIP. The latest version of SCIP can interrupt solving appropriately while LP is running. In this version of FiberSCIP, all `ParaSolver` threads are joined and FiberSCIP is terminated cleanly.

9.2 Time Limit Feature Implementation of FiberSCIP

In the previous versions of FiberSCIP, when a time limit is specified in the parameter, FiberSCIP created a `ParaTimeLimitMonitor` thread to create the time limit notification message to the `ParaLoadCoordinator`. The thread sleeps until the time limit, wakes up when time limit is reached, and sends the notification message to the `ParaLoadCoordinator`. The `ParaLoadCoordinator` tries to interrupt all `ParaSolver` threads. However, these interruptions within a reasonable time could be failed when LP is running within SCIP, since it did not have a chance to receive the message. From the performance point of view, creating the `ParaTimeLimitMonitor` is not good, but `ParaLoadCoordinator` works as a kind of event-driven controller, and then an event to notify the time limit was needed.

With UG version 1.0, this mechanism was changed to set a time limit when each SCIP solves a sub-MIP and hopes to detect the time limit on the `ParaSolver` side. How it works well depends on how accurately SCIP can terminate for the time limit setting. Unfortunately, it has several irregular timings and FiberSCIP needs to handle such cases currently. However, from the performance point of view, it has a benefit of running FiberSCIP without the `ParaTimeLimitMonitor` thread.

9.3 SelfSplit Ramp-up

In the general terminology of parallel branch-and-bound, ramp-up is a part of the initialization phase of the computation. The initialization phase is summarized by Henrich [46]. The *enumerative initialization* broadcasts the root node to all processes, which then perform an initial tree search according to the sequential algorithm. When the number of leaf nodes on each processor is at least the number of processes, processes can stop expanding. The i^{th} process then keeps the i^{th} node and deletes the rest. In this method, all processes are working from the very beginning and no communication is required. The *SelfSplit* [27] is refined in the enumerative initialization phase so that the open nodes are ordered more accurately to each solver that has a similar amount of work and so that a parallel MILP solver can perform static load balancing more efficiently. For QapNB parallelization, the enumeration initialization has a benefit, see Fujii et al. [29]. In UG version 1.0, the initialization method is implemented in B&B base classes as shown in Figure 5 so that any "base solver" can use it and name it SelfSplit ramp-up, that is, in the case of FiberSCIP, it is not limited to MILP, but also works for MINLP. Unlike in the original SelfSplit, it is used as a ramp-up and can perform dynamic load balancing after ramp-up, also, in order to cooperate with the other features of UG, it does layered presolving for sub-MIPs generated by SelfSplit and the sub-MIPs are checkpointing. FiberSCIP and ParaSCIP included in this release have this SelfSplit ramp-up feature.

9.4 Memory usage estimation

To make FiberSCIP stable, one of the features needed is handling the memory limit, since memory overuse makes FiberSCIP abort. However, memory usage estimation is very hard for FiberSCIP. By using SCIP functions for memory usage estimation (plus Linux system feature of memory usage, in case of FiberSCIP runs on Linux), the FiberSCIP memory usage estimation feature is implemented. When the estimation is more than the system memory, the latest version of FiberSCIP terminates with "memory limit reached".

10 The GCG Decomposition Solver

SCIP allows to implement decomposition-based algorithms within its framework. GCG is an extension that turns SCIP into a generic decomposition-based solver for MILPs. While GCG's focus is on Dantzig-Wolfe reformulation (DWR) and Lagrangian decomposition, Benders decomposition (BD) is also supported. The philosophy behind GCG is that decomposition-based algorithms like branch-price-and-cut (BP&C) can be routinely applied to MILPs without the user's interaction or even knowledge, just like branch-and-cut. To this end, GCG automatically detects a model structure that admits a decomposition, and then performs the corresponding reformulation. This results in a master problem and one or several subproblems, which are usually formulated as MILP problems. The latter are solved as sub-SCIPs or using specialized solvers. Based on the reformulation, the linear relaxation in every node is solved by column generation (in the DWR case), respectively, Benders cut generation (in the BD case). GCG features primal heuristics and separation of cutting planes, several of which are adapted from SCIP, but some are tailored to the decomposition situation in which both, an original and a reformulated model are available.

Since the last major release 3.0 in 2018, most development efforts went into improving the usability for experts and beginners alike. Besides few algorithmic features and code refactoring, the new release comes with more interfaces, a much improved documentation, and a collection of tools to support computational experiments and even more their evaluation/visualization. Even though not visible to the user, we improved the development process, in particular by establishing continuous integration pipelines etc. We like to think of GCG 3.5 as (the first half of) an *ecosystem release*.

10.1 Detection Loop Refactoring

Decomposition-based algorithms rely on model structures, such as a block-angular constraint matrix. For an automatic identification of such structures, GCG features a modular detection loop, which was introduced with version 3.0. So-called *detectors* iteratively assign roles like "master" or "block" to variables and/or constraints, potentially only to subsets, and possibly in several rounds. This way, usually many different potential decompositions are found. We refer to the SCIP Optimization Suite 6.0 release report [39] for a more detailed overview. Detectors are implemented as plugins such that new ones can be added conveniently. In every round, each detector works on existing (but possibly empty) partial or finished decompositions. An empirically very successful detection concept builds on the classification of constraints and variables, which is performed prior to the actual detection process, using so-called *classifiers*. Classifiers group variables or constraints according to arbitrary decomposition-relevant information, like type or name. Detectors then may or may not use the resulting classes to create (partial) decompositions.

GCG 3.5 comes with a refactored detection loop. Large parts of the code base were rewritten, moved, or renamed, while the general idea and procedure of the modular detection loop were maintained. Consequently, the interface changed significantly. First, (partial) decompositions, which were propagated, finished, or post-processed by the detectors, are represented by objects of the class `PARTIALDECOMP` (formerly `Seeed`). Objects of the class `DETPROBDATA` (formerly `Seeedpool`) manage all decompositions that were created during a run of the detection loop. Since the class `DETPROBDATA` should only manage the partial decompositions, many functions of `Seeedpool` were moved. Formerly, `Seeedpool` provided the functionality to classify constraints and variables. With version 3.5, variable and constraint classifiers are implemented as plugins as well. All classifiers are called at the beginning of the detection process. Each classifier can produce partitions of constraints or variables, which are represented by

objects of the classes `ConsPartition` (formerly `ConsClassifier`) and `VarPartition` (formerly `VarClassifier`), respectively. Both classes implement the abstract class `IndexPartition` (formerly `IndexClassifier`). This modular design allows users to easily add classifiers.

Furthermore, many parameters were changed. For a complete overview we refer to the `CHANGELOG`. Users can enable or disable the entire detection process using the parameter `detection/enabled`. Moreover, the parameter `detection/postprocess` enables or disables the post-processing of decompositions. Parameters related to classification were moved to `detection/classification/`. By default, GCG 3.5 preprocesses an instance, runs the detection, then solves. If detection should run already on the un-preprocessed model, it must be initiated manually before presolve starts; by default, a second round of detection is then still performed on the preprocessed model. With this more intuitive behavior, the parameters `origenabled` of detectors and classifiers were removed.

The legacy detection mode (for detectors prior to version 3.0) is no longer available. All corresponding parameters and legacy detectors were removed. Users have to implement detectors using the new callbacks and API.

10.2 Strong Branching in Branch-and-Price

In GCG, two general branching rules are implemented (branching on original variables [116] and Vanderbeck's generic branching [113]) as well as one rule that applies only to set partitioning master problems (Ryan and Foster branching [93]). While these rules differ quite significantly (creating two child nodes vs. several child nodes; branching on variables vs. on constraints), the general procedure at a node comes in two common stages: First, one determines the set of candidates we could possibly branch on (called the branching rule here). Second, the *branching candidate selection heuristic* then actually selects one of the available candidates. In SCIP the latter is done by ranking the candidates according to a score. Both the branching rule and the selection heuristic can have a significant impact on the size of the branch-and-bound tree, and hence on the runtime of the entire algorithm. GCG previously contained only pseudo cost, most fractional, and random branching as selection heuristics for original variable branching, and first-index branching for Ryan-Foster and Vanderbeck's generic branching. In GCG 3.5, several new selection heuristics are added, all of which are based on strong branching. For an overview of which selection heuristics are available for which branching rules see Table 6. In the following, we briefly describe the new selection heuristics. For more detailed descriptions, we refer to [37].

10.2.1 Branching Candidate Selection Heuristics Background

Given a set of branching candidates, the selection heuristic usually creates a ranking and selects a winner. One ranking criterion is the expected gain, that is, the improvement in dual bound in the child nodes when compared to the current node. However, computing the exact gains amounts to performing *all full strong branching*. In a branch-and-price context this means evaluating all branching candidates by solving all child node LP relaxations with column generation to optimality. With often hard pricing problems, this variant is an even (much) larger computational burden than it is in the standard branch-and-cut context. Yet, strong branching has demonstrated potential in branch-and-price for hard instances [83, 92]. In particular, strong branching generally creates small trees (compared to other branching rules).

To alleviate the computational effort, relaxations of strong branching are considered, such as not evaluating all candidates, or not solving the LP relaxations to optimality. In branch-and-price, one has even more degrees of freedom. In particular, we can choose to

Table 6: Branching candidate selection heuristics available in GCG 3.5.

branching rule / selection heuristic	original	Ryan-Foster	Vanderbeck
random/index-based branching	✓	✓	✓
most fractional/infeasible branching	✓	✓[2]	
pseudocost branching	✓	✓[2]	
strong branching with column generation[1]	✓	✓	
strong branching without column generation[1]	✓	✓	
hybrid branching[1]	✓	✓[3]	
reliability branching[1]	✓	✓[3]	
hierarchical branching[1]	✓	✓[3]	

[1]The strong branching based heuristics can be combined. [2]GCG can only aggregate the respective scores of the (two) individual variables. [3]These heuristics originally use both strong and pseudocost branching; however, pseudocost branching can also be substituted by any other heuristic, with varying performance.

not perform column generation when evaluating the candidates. Thus, we differentiate between *strong branching without column generation* (SBw/oCG) and *strong branching with column generation* (SBw/CG). In principle, one has the entire spectrum in between (partial or heuristic pricing, etc.) but this is beyond the scope here.

Another alternative to exact computations are *gain predictions*. A classical approach is *pseudocost branching*. Pseudocosts measure the average gain per eliminated fractionality for down and up branch separately, with the average being calculated from candidates which we branched on in the past that correspond to the same variable. Pseudocosts can be calculated very fast, however, using them also creates larger trees than strong branching. In particular, pseudocost scores are unreliable at the top of the tree, as there is only little/no historic data from which the averages can be calculated.

To combine the strengths and reduce the weaknesses of strong branching and pseudocost branching, they can also be used together. The three options for this that were added to GCG 3.5 are *hybrid strong/pseudocost branching* [63], *reliability branching* [4], and *hierarchical strong branching* [37, 83, 92]. In hybrid strong/pseudocost branching, strong branching is applied only for nodes up to a given depth, and pseudocost branching to the rest.

For reliability branching, each candidate is assigned a reliability score, which is simply the minimum of the number of down and up branches in the tree of candidates corresponding to the same variable. The reliability score of a candidate reflects how close the pseudocost score for the candidate is likely to be to the actual gain, that is, how *reliable* the prediction is. Candidates whose reliability score is below a certain threshold are evaluated with strong branching, while the remaining candidates are again evaluated using pseudocosts.

For hierarchical strong branching, the selection process is divided into three phases (also called a hierarchy in the literature), where

- in phase 0, the candidates are filtered based on some heuristic that is quick to compute (such as pseudocost, most fractional, or random branching), then
- in phase 1, the remaining candidates are filtered based on their SBw/oCG scores, and finally
- in phase 2, a candidate is selected out of the remaining candidates based on the score

the candidates received from SBw/CG.

The effort at a given node depends on the assumed importance of evaluating the node precisely (a larger estimated size of the subtree gives more importance to a candidate), and on the difference in computational effort vs. quality of predictions between the phases. The intuition here is that only the most promising candidates (based on the scores from the earlier heuristics) should receive the largest evaluation effort (and best evaluation quality).

In the same way that strong branching can be combined with pseudocost branching to obtain hybrid strong/pseudocost branching and reliability branching, we can also combine hierarchical strong branching with hybrid strong/pseudocost branching and reliability branching to obtain *hybrid hierarchical strong/pseudocost branching* and *hierarchical reliability branching* [37]: for hierarchical reliability branching, we only perform strong branching in phase 1 and 2 on candidates that are not yet reliable, with different thresholds for phase 1 and phase 2. For hybrid hierarchical strong/pseudocost branching, we only perform phase 0 starting from a given depth, and phases 1 and 2 up to a given depth (again separate thresholds for each phase).

Strong branching is implemented using SCIP's probing mode. The columns generated when evaluating a node using SBw/CG are kept in GCG's column pool. The (potentially positive side) effect of this needs still to be evaluated.

10.2.2 Parameters for Strong Branching

By default, like in SCIP, strong branching is disabled. It can be enabled for original variable branching or Ryan-Foster branching by setting `branching/orig/usestrong` or `branching/ryanfoster/usestrong`, respectively, to `TRUE`. By default, this performs a hybrid hierarchical branching. Furthermore, there are several parameters that allow to completely change the behavior of the heuristics. In fact, all of the strong branching heuristics use the same implementation, just with different parameter settings. Preset settings files for each of the previously described selection heuristics as well as a template file can be found in GCG's `settings` folder. Table 7 lists most of the available parameters. Further information can be found in the paper by Gaul [37] and in GCG's documentation.

10.2.3 Performance Evaluation

A preliminary performance comparison (for original variable branching only) regarding the number of nodes and computation times can be found in the tables in Appendix B and in Figure 6. The computations were performed using an Intel Xeon L5630 Quad Core with 2.13 GHz and 16 GB of RAM. The time limit was set to 3600 seconds. All instances in our testset are taken from the unreleased strIPlib (`striplib.or.rwth-aachen.de`). Instances contained in strIPlib are all known to contain a model structure to which a DWR is applicable. We selected instances that need at least 1000 nodes to solve (with a previous GCG version). Diversity of the testset was controlled in a manner similar to the creation of the MIPLIB 2017 benchmark set [40]. We refer to Gaul [37] for further information about the experiments.

The preliminary results suggest that it is conceivable to have strong branching components enabled in branch-and-price by default. A potential explanation is that solving the subproblems is very costly, and processing a node is more expensive than in standard branch-and-bound; thus there is an ever stronger incentive to have a smaller tree in branch-and-price. Certainly, this needs more investigation.

Table 7: Parameters for strong branching.

Parameter	Effect
`branching/[orig,ryanfoster]/...`	
`minphase[0,1]outcands`	minimum number of output candidates from phase [0,1]
`maxphase[0,1]outcands`	maximum number of output candidates from phase [0,1]
`maxphase[0,1]outcandsfrac`	maximum number of output candidates from phase 0 as fraction of total candidates, takes precedence over `minphase[0,1]outcands`
`phase[1,2]gapweight`	how much influence the nodegap has on the number of output candidates from phase [1,2]−1
`branching/bpstrong/...`	
`histweight`	fraction of candidates in phase 0 that are chosen based on historical strong branching performance
`mincolgencands`	minimum number of candidates for phase 2 to be performed, otherwise the best previous candidate will be chosen
`maxsblpiters`, `maxsbpricerounds`	upper bound on number of simplex iterations/pricing rounds, sets upper bound to twice the average if set to 0
`immediateinf`	if set to `TRUE`, candidates with infeasible children are selected immediately
`reevalage`	reevaluation age
`maxlookahead`	upper bound for the look ahead
`lookaheadscales`	by how much the look ahead scales with the overall evaluation effort (currently `lookaheadscales` * `maxlookahead` is the minimum look ahead)
`closepercentage`	fraction of the chosen candidate's phase 2 score the phase 0 heuristic's choice needs to have in order to be considered close
`maxconsecheurclose`	number of times in a row the phase 0 heuristic needs to be close for strong branching to be stopped entirely
`minphase0depth`, `maxphase[1,2]depth`, `depthlogweight`, `depthlogbase`, `depthlogphase[0,2]frac`	$\kappa_1^+ = \lambda^1_{\text{depth}} + \rho_{\text{depth}} \cdot \log_{\lambda_{\text{base}}}(n_{\text{cands}})$, where κ_i^+ is the depth until which phase i is performed, λ^1_{depth} = `maxphase1depth`, ρ_{depth} = `depthlogweight`, λ_{base} = `depthlogbase` and n_{cands} is number of variables that we could branch on (usually all integer and binary variables). $\kappa_2^+ = \kappa_1^+ \cdot$ `depthlogphase2frac`, but at most `maxphase2depth`. The minimum depth from which on phase 0 is performed is equal to $\kappa_1^+ \cdot$ `depthlogphase0frac`, but at least `minphase0depth`
`phase[1,2]reliable`	min count of pseudocost scores for a variable to be considered reliable in phase [1,2]

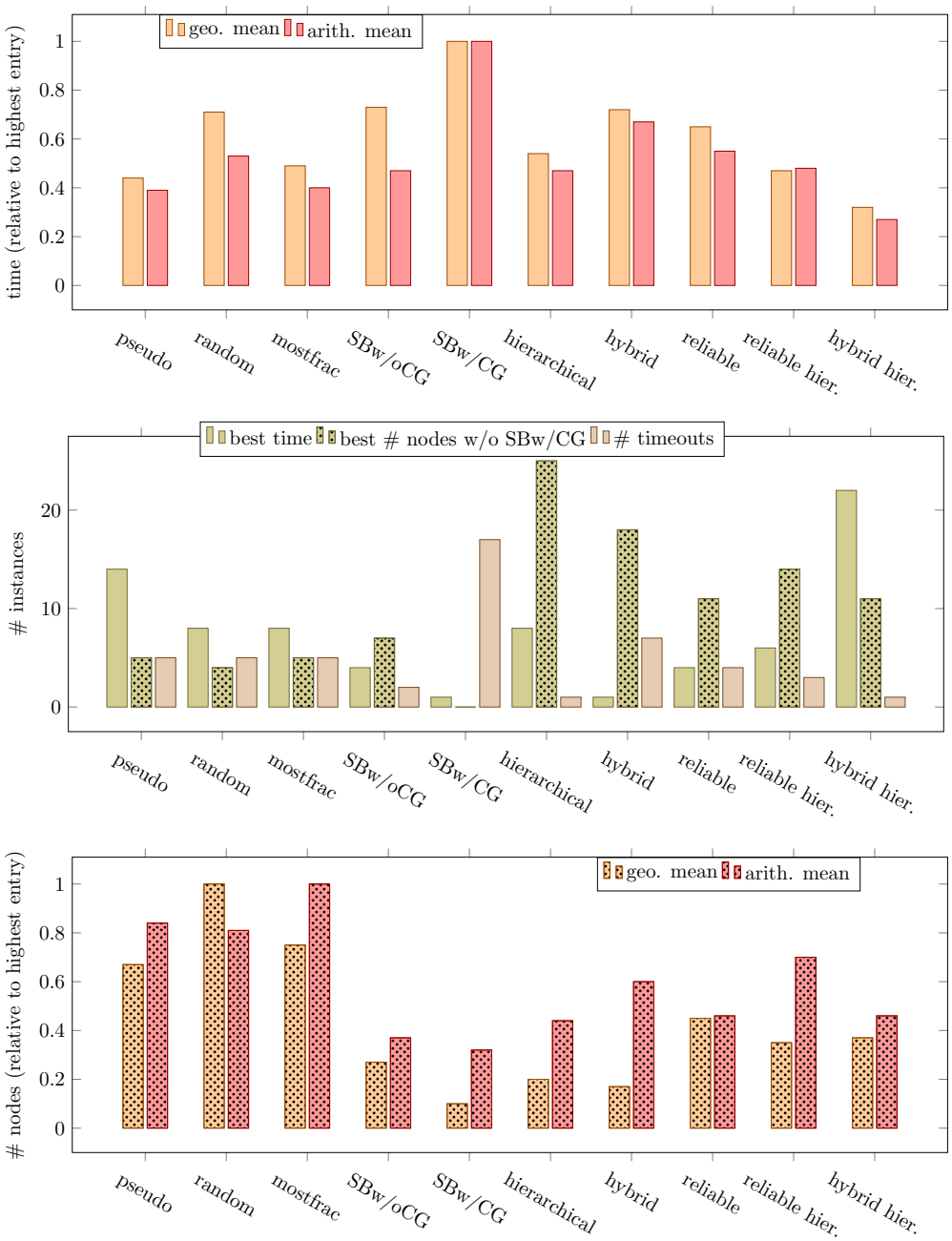

Figure 6: Visualizations for original variable branching based on tables from Appendix B. The upper plot shows the geometric and arithmetic mean for the amount of time needed relative to the highest in each category. The lower plot shows the same for the number of nodes needed. The middle figure shows the number of timeouts for each heuristic, and the number of times a heuristic was the best for a given instance regarding nodes (excluding full strong branching) and time.

10.3 Python Interface

With GCG 3.5 we introduce PyGCGOpt which extends SCIP's existing Python interface [67] for GCG. It is implemented in Cython (`cython.org`) and is distributed as a package independent from the optimization suite under `github.com/scipopt/PyGCGOpt`. All the existing functionality for the modeling of MILPs is inherited from PySCIPOpt. As a result, any MILP modeled in Python can also be solved with GCG without additional effort. This lowers the technical hurdle to try out a branch-and-price approach for any existing problem. In its first incarnation, the interface supports specifying custom decompositions and exploration of automatically detected decompositions. They can be visualized directly within Jupyter notebooks. In addition, GCG plugins for `detectors` and `pricing solvers` can be implemented in Python.

In the following code listing, the capacitated p-median problem (CPMP) is modeled with PySCIPOpt's expression syntax. The specified textbook decomposition [19] is solved by GCG with Dantzig-Wolfe reformulation upon the call to `m.optimize()`. Note that the automatic structure detection functionality of GCG remains intact, so that the user does not need to (but can) specify a decomposition.

```
from pygcgopt import gcgModel, quicksum as qs

n_locs = 5
n_clusters = 2
distances = {0: {0: 0, 1: 6, 2: 54, 3: 52, 4: 19}, 1: {0: 6, 1: 0, 2: 28,
    3: 75, 4: 61}, 2: {0: 54, 1: 28, 2: 0, 3: 91, 4: 40}, 3: {0: 52, 1:
    75, 2: 91, 3: 0, 4: 28}, 4: {0: 19, 1: 61, 2: 40, 3: 28, 4: 0}}
demands = {0: 14, 1: 13, 2: 9, 3: 15, 4: 6}
capacities = {0: 39, 1: 39, 2: 39, 3: 39, 4: 39}

m = gcgModel()
x = {(i, j): m.addVar(f"x_{i}_{j}", vtype="B", obj=distances[i][j]) for i
    in range(n_locs) for j in range(n_locs)}
y = {j: m.addVar(f"y_{j}", vtype="B") for j in range(n_locs)}

conss_assignment = m.addConss(
  [qs(x[i, j] for j in range(n_locs)) == 1 for i in range(n_locs)])
conss_capacity = m.addConss(
  [qs(demands[i] * x[i, j] for i in range(n_locs)) <= capacities[j] * y[j]
     for j in range(n_locs)])
cons_pmedian = m.addCons(qs(y[j] for j in range(n_locs)) == n_clusters)

master_conss = conss_assignment + [cons_pmedian]
block_conss = [[cons] forimize()
```

The Python interface required refactoring within the codebase of GCG. Before, a lot of core functionality of the solver was implemented within dialog handlers. This made it hard to use GCG as a library in external programs. The functions `gcgtransformProb()`, `gcgpresolve()`, `gcgdetect()`, `gcgsolve()`, `gcggetDualbound()`, `gcggetPrimalbound()`, and `gcggetGap()` were added to the public interface and are called from the dialog handlers as well as the Python interface. As a side effect, GCG can now be used better as a C/C++ shared library.

10.4 Visualization Suite

Visualizations of algorithmic behavior can yield understanding and intuition for interesting parts of a solving process. With GCG 3.5, we include a *visualization suite* that offers different visualization scripts to show processes and results related to detection, branching, or pricing, among others. These scripts are written in Python 3 and included in the folder `stats` and use the `.out`, `.res` and `.vbc` files generated when executing `make test STATISTICS=true` (possible additional requirements are given in the documentation).

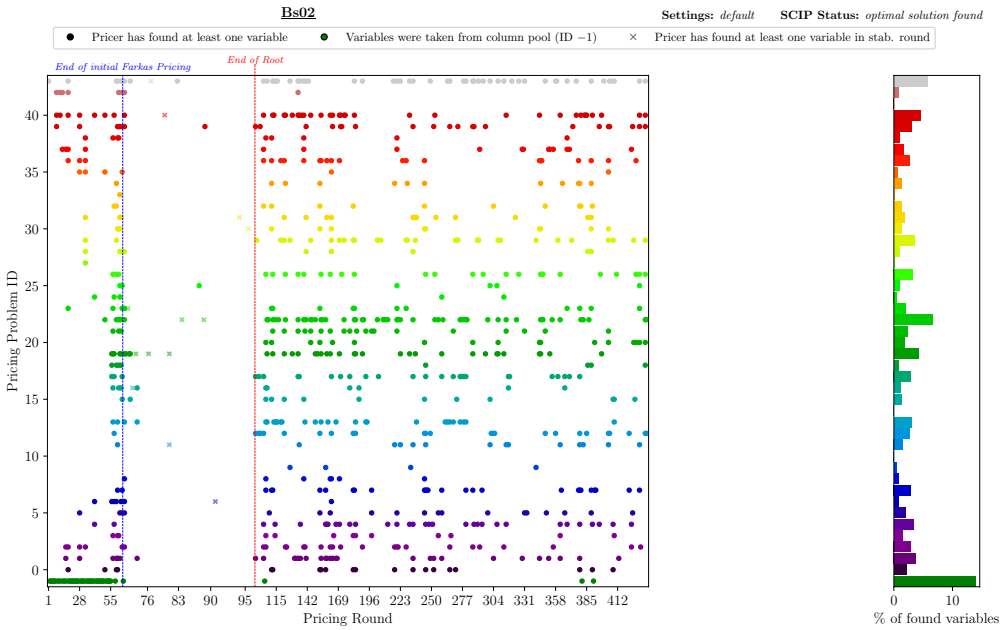

Figure 7: Bubble plot visualizing how the pricing problems performed during GCG's Branch-and-Price process. This visualization was automatically generated using the new comparison report functionality.

Furthermore, the suite also allows for two additional ways of accessing the visualization scripts:

1. Reporting functionality: With two different scripts, callable via `make visu`, users can easily generate reports similar to the *decomposition report* that was already available in GCG 3.0, which offers an overview over all decompositions that GCG found during its detection process. The generated documents include all visualizations offered by the suite along with descriptions of them in the captions. While the *testset report* shows information about a single run of one selected testset, the *comparison report* also compares two or more runs. Examples of both reports can now be found in the GCG website documentation, see Section 10.5.
2. Jupyter notebook: Since the scripts themselves already require a working installation of Python 3, we now added a *visualization notebook* with which one can read data (sample data provided in the GCG website documentation), clean, and filter it interactively, and visualize the results afterwards. The scripts of the visualization suite are imported and returned plots can be shown, exported, and even further edited.

Just like GCG should facilitate experimenting with a decomposition approach without having to implement it, the visualization suite should facilitate producing and presenting computational results and algorithmic behavior. Also this is an ongoing long term effort.

10.5 Website Documentation

The online documentation of GCG was lagging behind the progress made with the code itself. As part of this release, we offer a user-group targeted website documentation. It enables users to make themselves familiar with GCG by means of very accessible

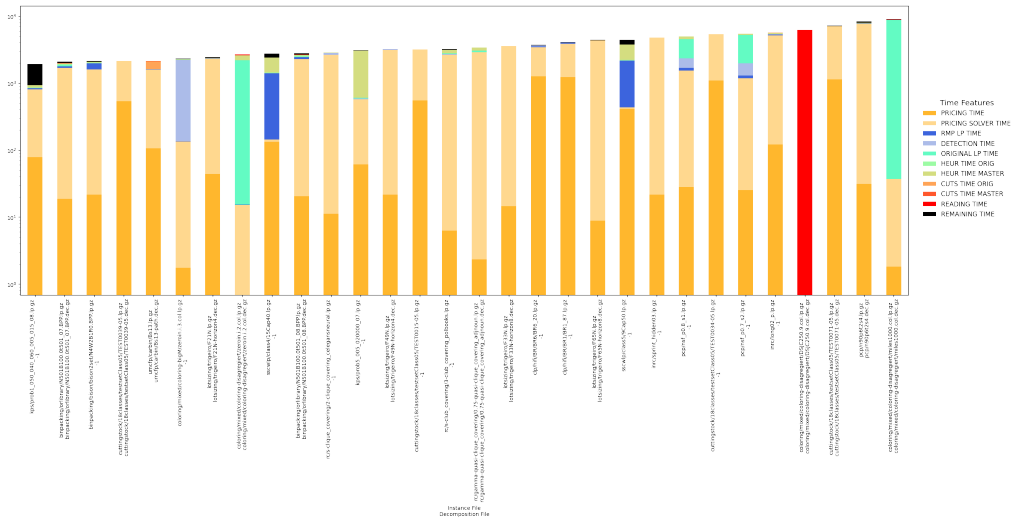

Figure 8: Time Distribution of a set of randomly drawn samples from the strIPlib. This visualization was generated using the included Jupyter notebook.

feature descriptions for functionality such as the explore menu or the visualization suite and by a set of use cases to follow and reproduce. For developers, we now include a guide explaining the peculiarities of the interplay between GCG and SCIP ("Getting Started: Developer's Edition"). Within the "Developer's Guide", descriptions of existing code and algorithmics such as detection, branching, and pricing are provided to allow developers to familiarize themselves with them, if required. Updates to the "How to use" (for instance, conducting experiments) and "How to add" (for instance, adding branching rules) sections completes the documentation.

11 SCIP-SDP

SCIP-SDP is a framework for solving mixed-integer semidefinite programs of the following form

$$\begin{aligned} \inf \quad & b^\top y \\ \text{s.t.} \quad & \sum_{k=1}^{m} A^k y_k - A^0 \succeq 0, \\ & \ell_i \leq y_i \leq u_i \qquad \forall\, i \in [m], \\ & y_i \in \mathbb{Z} \qquad \forall\, i \in I, \end{aligned} \tag{37}$$

with symmetric matrices $A^k \in \mathbb{R}^{n\times n}$ for $i \in \{0,\dots,m\}$, $b \in \mathbb{R}^m$, $\ell_i \in \mathbb{R}\cup\{-\infty\}$, $u_i \in \mathbb{R}\cup\{\infty\}$ for all $i \in [m] \coloneqq \{1,\dots,m\}$. The set of indices of integer variables is given by $I \subseteq [m]$ and $M \succeq 0$ denotes that a matrix M is positive semidefinite.

SCIP-SDP was initiated by Sonja Mars and Lars Schewe, see Mars [74], and then continued by Gally et al. [32] and Gally [30]. It features interfaces to the SDP-solvers DSDP, Mosek, and SDPA. In the following, we briefly report on the changes since the last version 3.2.0.

SCIP-SDP 4.0 contains about 50 000 lines of C-code. Since the last version most of these lines have been touched. In particular, the interface to the SDP-solvers has been completely revised. One benefit is that the memory footprint of SCIP-SDP is now smaller for large instances. Moreover, many bugs have been fixed.

Two important parameter changes that impact performance are:

– By default the number of used threads is 1 (it was previously set to "automatic"). This change speeds-up the solution process by about 40 % for most smaller to medium sized SDPs.
– The feasibility and optimality tolerances have been set to 10^{-5}. The exception is Mosek for which is is set to 10^{-6}, because this leads to more reliable results.

Further changes in a nutshell are the following:

– If the SDP-relaxation only has a single variable, it is solved using a semismooth Newton method. This slightly speeds up solution times and significantly decreases the times for heuristics. In particular, this holds for rounding heuristics on instances in which all integral variables are fixed except a single continuous variable. This continuous variable is often used for expressing the objective function, for example in cardinality least squares problems.
– The LP-rows that are added to the LP-relaxation are strengthened using standard LP-preprocessing routines (coefficient tightening).
– A new heuristic `heur_fracround` has been added which iteratively rounds integer variables based on their fractional values in the last SDP-relaxation. In between, it performs propagation and solves a final SDP if unfixed continuous variables remain. This heuristic helps to significantly improve the overall running times. Propagation is now also used by the heuristic `heur_sdprand` to improve its success rate for instances with additional linear constraints. Furthermore, both heuristics are correcting nearly integral values of integral variables in order to avoid small rounding errors, which might add up to significant amounts.
– Several new presolving techniques have been introduced, which are discussed and evaluated in detail by Matter and Pfetsch [75]. This includes two propagation methods to fix variables based on 2×2-minors and the upper bounds of other variables.
– SCIP-SDP also allows to use LP-solving instead of SDP-relaxations using the parameter `misc/solvesdps`. It then generates so-called eigenvector cuts. The behavior of these cuts has been changed as follows. One can now add eigenvector cuts for all negative eigenvalues of the current infeasible relaxation. Moreover, SDP-relaxations can be solved in enforcing, that is, after all integer variables have integral values. Furthermore, the cuts can be sparsified.
– The display of SCIP-SDP now changes depending on whether SDPs or LPs are solved for the relaxations. Moreover, the default settings are redefined for solving SDPs.
– One can also generate a second-order cone relaxation, but so far this has not shown a run time improvement.
– The readers for the SDPA and CBF formats have been completely revised (and rewritten for SDPA). They are now much faster and produce more warnings if errors occur.
– SCIP-SDP can also handle rank-1 constraints, that is, the requirement that the resulting matrix has rank 1. This is achieved by adding quadratic constraints for 2×2-minors. Rank-1 constraints regularly appear in the literature, but are usually very hard to solve. The handling of these constraints has been revised.
– The locking information (capturing whether the matrices A^k are positive/negative semidefinite) is now copied to sub-SCIPs.
– The statistics for solving SDP-relaxations has been extended and now reports more details.
– There is a new file `scipsdpdef.h` that contains defines for the SCIP-SDP version. This enables code to depend on different SCIP-SDP versions.

Table 8: Performance comparison of SCIP-SDP 4.0 vs. SCIP-SDP 3.2

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

– It is now possible to add SDP-constraints within the solving process.

– SCIP-SDP can now run concurrently, for example, by writing `concurrentopt` in the command line if SCIP and SCIP-SDP are compiled using the `TPI`.

– The updated Matlab Interface presented in Section 7 also allows to use SCIP-SDP.

Before we present some computational results, let us add some words of caution. Although SCIP-SDP is numerically quite robust, accurately solving SDPs is more demanding than solving LPs. This can lead to wrong results on some instances[11] and the results often depend on the chosen tolerances. Technical reasons are that the SDPs are solved using interior point solvers, which produce solutions with more "numerical noise" (since they do not have nonbasic variables). Moreover, the solvers use relative tolerances, while SCIP-SDP uses absolute tolerances. Finally, for Mosek, we use a slightly tighter tolerance than in SCIP-SDP.

Table 8 shows a comparison between SCIP-SDP 3.2 and 4.0 on the same testset as used by Gally et al. [32], which consists of 194 instances. Reported are the number of optimally solved instances, as well as the shifted geometric means of the number of processed nodes and the CPU time in seconds. We use Mosek 9.2.40 for solving the continuous SDP-relaxations. The tests were performed on a Linux cluster with 3.5 GHz Intel Xeon E5-1620 Quad-Core CPUs, having 32 GB main memory and 10 MB cache. All computations were run single-threaded and with a timelimit of one hour.

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

12 SCIP-Jack: Solving Steiner Tree and Related Problems

Given an undirected, connected graph $G = (V, E)$, costs (or weights) $c : E \to \mathbb{R}_+$ and a set $T \subseteq V$ of *terminals*, the *Steiner tree problem in graphs* (SPG) asks for a tree $S = (V(S), E(S)) \subseteq G$ such that $T \subseteq V(S)$ holds and $\sum_{e \in E(S)} c(e)$ is minimized. The SPG is a fundamental $\mathcal{NP}$-hard problem [54], and one of the most studied problems in combinatorial optimization. Moreover, many related problems have been extensively described in the literature and can be found in a wide range of practical applications [64].

Since version 3.2, the SCIP Optimization Suite has contained SCIP-Jack, an exact solver not only for the SPG but also for 11 related problems. This release of the SCIP Optimization Suite contains the new SCIP-Jack 2.0[12], which can handle two additional problem classes: the maximum-weight connected subgraph problem with budgets, and the partial-terminal Steiner tree problem. Furthermore, SCIP-Jack 2.0 comes with major improvements on almost all problem classes it can handle. Most importantly, the latest SCIP-Jack outperforms the well-known SPG solver by Polzin and Vahdati [84, 112] on almost all nontrivial benchmark testsets from the literature. See the preprint [88] for more details. Notably, Polzin and Vahdati [84, 112] had remained out of reach for almost 20 years for any other SPG solver.

[11] For instance, in seldom cases, the dual bound might exceed the value of a primal feasible solution.
[12] see also `http://scipjack.zib.de`

The large number of newly implemented algorithms (and data structures) also results in an increase of the SCIP-Jack code base by a factor of almost 3: To roughly 110 000 lines of code. Additionally, the implementation of many existing methods has been improved. We list several of the most important new features.

For SPG, a central new feature is a distance concept that provably dominates the well-known bottleneck Steiner distance from Duin and Volgenant [25], see Rehfeldt and Koch [90] for details. This distance concept is used in several (new) reduction methods implemented in SCIP-Jack 2.0. Also, the new SCIP-Jack includes a full-fledged implementation of so-called *extended reduction techniques*. These methods are provably stronger than the state-of-the-art implementation [85], and also yield strong practical results, see Rehfeldt and Koch [88] for details. Furthermore, decomposition methods for the SPG have been implemented, for example to exploit the existence of biconnected components in the underlying graph. Also, dynamic programming algorithms have been implemented to efficiently solve subproblems with special structures that sometimes arise after decomposition.

The improvements for SPG also have an immediate impact on problems that are transformed to SPG within SCIP-Jack, such as the group Steiner tree problem. Even for the Euclidean Steiner tree problem, large improvements are possible (the SPG can be used for full Steiner tree concatenation after the discretization of the problem): SCIP-Jack 2.0 is able to solve 19 Euclidean Steiner tree problems with up to 100 000 terminals for the first time to optimality, see Rehfeldt and Koch [88]. Notably, the state-of-the-art Euclidean Steiner tree solver GeoSteiner 5.1 [49] could not solve any of these instances even after one week of computation. In contrast, SCIP-Jack 2.0 solves all of them within 12 minutes, some even within two minutes.

Considerable problem-specific improvements have also been made for the prize-collecting Steiner tree problem and (to a lesser extent) for the maximum-weight connect subgraph problem. For details on the improvements for the prize-collecting Steiner tree problem see Rehfeldt and Koch [89], for the maximum-weight connect subgraph problem see Rehfeldt, Franz, and Koch [91]. The improvements encompass primal and dual heuristics as well as reduction techniques. As a result, SCIP-Jack 2.0 can solve many previously unsolved benchmark instances from both problem classes to optimality—the largest of these instances have up to 10 million edges. Additionally, for the prize-collecting Steiner tree problem SCIP-Jack 2.0 can solve most benchmark sets from the literature more than two times faster than its predecessor with respect to the shifted geometric mean (with a shift of 1 second).

13 Final Remarks

The SCIP Optimization Suite 8.0 release provides new functionality and improved performance and reliability. In SCIP, new symmetry handling features were added, including the handling of symmetries of general integer and continuous variables, improving detection routines and adding a strategy for symmetry handling routine selection. Mixing cutting planes were implemented, which considerably improve the performance on chance constrained programs. A decomposition primal heuristic was updated to further improve the found solutions, and a new decomposition primal heuristic was added. A new cut strengthening procedure was added to the Benders decomposition framework, and a new type of plugin for cut selection was introduced.

With this release also comes a thorough revision of how nonlinear constraints are handled in SCIP, in particular how extended formulations are created. In the new version, the original formulation is preserved and the extended formulation is used for relaxations only, which drastically improves the reliability of solutions. High- and low-level nonlinear structures are now handled by plugins of different types in order to avoid expression type ambiguity. Simultaneously, a number of new MINLP features were introduced such as

various new cutting planes, symmetry detection for nonlinear constraints, support for sine and cosine functions, and others.

Regarding usability, the Julia package SCIP.jl was improved in several aspects and a new MATLAB interface to SCIP was implemented. UG was generalized to enable the parallelization of all solvers via a unified framework, without the need to modify the framework for each solver; its internal structure has been completely reworked. The new version of GCG includes new algorithmic features and substantial ecosystem improvements, such as extended interfaces, improved documentation, added utility for running and analyzing computational experiments. PaPILO features a new postsolving functionality for dual solutions when applied to pure LPs. The handling of relaxations in SCIP-SDP was revised, and new heuristics and presolving methods were added. The new version of SCIP-Jack can handle two additional problem classes and comes with major performance improvements.

These developments yield a considerable performance improvement on nonconvex MINLP instances and reduce the overall number of numerical failures on the MINLP testset, although a slowdown is observed on convex instances due to a lack of recognition of a structure present in one instance group. Nonetheless, we observe an overall runtime reduction of about 21%. A substantial speed-up is observed on MILP instances with about a 50% shorter runtime on the most challenging instances, as well as special problem classes such as SDP and Steiner tree problems and their variants.

Acknowledgements

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

Contributions of the Authors

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

JvD, CH, and MP are responsible for the changes of the symmetry handling routines (Section 3.2). The extension of symmetry handling to nonlinear constraints (Section 3.2.2) is by FW. WC and MP implemented the mixing cut separator (Section 3.3). The update of `PADM` and the new `DPS` heuristic (Section 3.4) are due to KH and DW. SJM is responsible for the updates to the Benders' decomposition framework (Section 3.5). MT and FeS implemented the new cut selector plugin (Section 3.6). Various technical improvements (Section 3.7) were added by MP and SV. The new expressions framework (Section 4.1) is by BM, FeS, FW, KB, and SV. The rewritten handler for nonlinear constraints (Section 4.2) is by BM, FeS, KB, and SV. The nonlinear handler for quadratic expressions (Section 4.3) and the separator `sepa_interminor` (Section 4.11) are by AC and FeS. The nonlinear handler for second-order cones (Section 4.4) is by BM, FeS, and FW. The nonlinear handler for bilinear expressions (Section 4.5) and the separator `sepa_minor` (Section 4.10) are by BM. The nonlinear handler for convex and concave expressions (Section 4.6) are by BM, KB, and SV. The nonlinear handler for quotients (Section 4.7) is by BM and FW. The nonlinear handler for perspective reformulations (Section 4.8) is by KB. The separator for RLT cuts (Section 4.9) is by FW and KB. The separator for principal minors of $X \succeq xx^\top$ (Section 4.10) is by BM and FW. The separator for

intersection cuts on the rank-1 constraint for a matrix (Section 4.11) is by AC and FeS. The revised primal heuristic `subnlp` (Section 4.12) and the updates to NLP, NLPI, and AD interfaces (Section 4.13) are by SV.

The changes to SoPlex (Section 5) are due to LE and AH. AH and AG are responsible for the new dual postsolving functionality in PaPILO (Section 6). The new AMPL interface of SCIP (Section 7.1) was implemented by SV. The Julia interface `SCIP.jl` (Section 7.2) was extended and updated by MB, Erik Tadewaldt, Robert Schwarz, and Yupei Qi. The SoPlex C interface (Section 7.3) was developed by AC, LE, and MB. Nicolai Simon and MP updated the Matlab-interface (Section 7.4). The work on ZIMPL (Section 8) was done by TK. The updates to the UG framework (Section 9) are by YS. Concerning GCG (Section 10), EM refactored the detector loop; the website documentation and visualization suite is due to TD; OG created the strong branching code; and SS implemented PyGCGOpt. FM and MP implemented the changes in SCIP-SDP (Section 11). DR is responsible for SCIP-Jack (Section 12).

The work by FrS on the continuous integration system, regular test and benchmark runs, binary distributions, websites, and many more has been invaluable for all developments.

References

[1] A. Abdi and R. Fukasawa. On the mixing set with a knapsack constraint. *Mathematical Programming*, 157:191–217, 2016. doi:10.1007/s10107-016-0979-5.

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

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

[4] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, 2005.

[5] T. Achterberg, R. E. Bixby, Z. Gu, E. Rothberg, and D. Weninger. Presolve reductions in mixed integer programming. *INFORMS Journal on Computing*, 32(2):473–506, 2020. doi:10.1287/ijoc.2018.0857.

[6] W. P. Adams and H. D. Sherali. A tight linearization and an algorithm for zero-one quadratic programming problems. *Management Science*, 32(10):1274–1290, 1986.

[7] W. P. Adams and H. D. Sherali. Linearization strategies for a class of zero-one mixed integer programming problems. *Operations Research*, 38(2):217–226, 1990.

[8] W. P. Adams and H. D. Sherali. Mixed-integer bilinear programming problems. *Mathematical Programming*, 59(1):279–305, 1993.

[9] A. Atamtürk, G. L. Nemhauser, and M. W. Savelsbergh. The mixed vertex packing problem. *Mathematical Programming*, 89:35–53, 2000. doi:10.1007/s101070000154.

[10] E. Balas. Intersection cuts—a new type of cutting planes for integer programming. *Operations Research*, 19:19–39, 1971.

[11] X. Bao, N. V. Sahinidis, and M. Tawarmalani. Multiterm polyhedral relaxations for nonconvex, quadratically-constrained quadratic programs. *Optimization Methods and Software*, 24(4-5):485–504, 2009. doi:10.1080/10556780902883184.

[12] P. Belotti, J. Lee, L. Liberti, F. Margot, and A. Wächter. Branching and bounds tightening techniques for non-convex MINLP. *Optimization Methods and Software*, 24(4-5):597–634, 2009. doi:10.1080/10556780903087124.

[13] P. Belotti, C. Kirches, S. Leyffer, J. Linderoth, J. Luedtke, and A. Mahajan. Mixed-integer nonlinear optimization. *Acta Numerica*, 22:1–131, 2013. doi:10.1017/S0962492913000032.

[14] P. Bendotti, P. Fouilhoux, and C. Rottner. Orbitopal fixing for the full (sub-)orbitope and application to the unit commitment problem. *Mathematical Programming*, 186:337–372, 2021. doi:10.1007/s10107-019-01457-1.

[15] T. Berthold and J. Witzig. Conflict analysis for minlp. *INFORMS Journal on Computing*, 33(2):421–435, 2021. doi:10.1287/ijoc.2020.1050.

[16] T. Berthold, S. Heinz, and M. E. Pfetsch. Nonlinear pseudo-boolean optimization: relaxation or propagation? In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing – SAT 2009*, number 5584 in LNCS, pages 441–446. Springer, 2009.

[17] K. Bestuzheva, A. Gleixner, and S. Vigerske. A computational study of perspective cuts. ZIB Report 21-07, Zuse Institute Berlin, 2021.

[18] R. Borndörfer, C. E. Ferreira, and A. Martin. Decomposing matrices into blocks. *SIAM Journal on Optimization*, 9(1):236–269, 1998. doi:10.1137/S1052623497318682.

[19] A. Ceselli and G. Righini. A branch-and-price algorithm for the capacitated p-median problem. *Networks*, 45(3):125–142, 5 2005. doi:10.1002/net.20059. URL `www.interscience.wiley.`

[20] J.-S. Chen and C.-H. Huang. A note on convexity of two signomial functions. *Journal of Nonlinear and Convex Analysis*, 10(3):429–435, 2009.

[21] A. Chmiela, G. Muñoz, and F. Serrano. On the implementation and strengthening of intersection cuts for QCQPs. *Integer Programming and Combinatorial Optimization Proceedings*, 22:134–147, 2021. doi:10.1007/978-3-030-73879-2_10.

[22] Concorde. Concorde tsp solver. `https://www.math.uwaterloo.ca/tsp/concorde.html`, 2021.

[23] S. S. Dey and M. Molinaro. Theoretical challenges towards cutting-plane selection. *Mathematical Programming*, 170(1):237–266, 2018.

[24] F. Domes and A. Neumaier. Constraint propagation on quadratic constraints. *Constraints*, 15(3):404–429, 2010. ISSN 1383-7133. doi:10.1007/s10601-009-9076-1.

[25] C. Duin and A. Volgenant. An edge elimination test for the Steiner problem in graphs. *Operations Research Letters*, 8(2):79–83, 1989. doi:10.1016/0167-6377(89)90005-9.

[26] M. A. Duran and I. E. Grossmann. An outer-approximation algorithm for a class of mixed-integer nonlinear programs. *Mathematical Programming*, 36(3):307–339, 1986. doi:10.1007/BF02592064.

[27] M. Fischetti, M. Monaci, and D. Salvagnin. Selfsplit parallelization for mixed-integer linear programming. *Computers and Operations Research*, 93:101–112, 2018. doi:https://doi.org/10.1016/j.cor.2018.01.011.

[28] A. Frangioni and C. Gentile. Perspective cuts for a class of convex 0–1 mixed integer programs. *Mathematical Programming*, 106(2):225–236, 2006.

[29] K. Fujii, N. Ito, S. Kim, M. Kojima, Y. Shinano, and K.-C. Toh. Solving challenging large scale qaps. Technical Report 21-02, ZIB, Takustr. 7, 14195 Berlin, 2021.

[30] T. Gally. *Computational Mixed-Integer Semidefinite Programming*. PhD thesis, TU Darmstadt, 2019.

[31] T. Gally, M. E. Pfetsch, and S. Ulbrich. A framework for solving mixed-integer semidefinite programs. *Optimization Methods and Software*, 33(3):594–632, 2017. doi:10.1080/10556788.2017.1322081.

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

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

[34] 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, Optimization Online, 2016. URL `http://www.optimization-online.org/DB_HTML/2016/03/5360.html`.

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

[36] G. Gamrath, D. Anderson, K. Bestuzheva, W.-K. Chen, L. Eifler, M. Gasse, P. Gemander, A. Gleixner, L. Gottwald, K. Halbig, G. Hendel, C. Hojny, T. Koch, P. L. Bodic, S. J. Maher, F. Matter, M. Miltenberger, E. Mühmer, B. Müller, M. E. Pfetsch, F. Schlösser, F. Serrano, Y. Shinano, C. Tawfik, S. Vigerske, F. Wegscheider, D. Weninger, and

J. Witzig. The SCIP Optimization Suite 7.0. Technical report, Optimization Online, 2020. http://www.optimization-online.org/DB_HTML/2020/03/7705.html.

[37] O. Gaul. Hierarchical strong branching and other strong branching-based branching candidate selection heuristics in branch-and-price. Master's thesis, RWTH Aachen University, 2021.

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

[39] A. Gleixner, M. Bastubbe, L. Eifler, T. Gally, G. Gamrath, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. E. Lübbecke, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, C. Schubert, F. Serrano, Y. Shinano, J. M. Viernickel, M. Walter, F. Wegscheider, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 6.0. Technical report, Optimization Online, 2018. URL http://www.optimization-online.org/DB_HTML/2018/07/6692.html.

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

[41] F. Glover. Convexity cuts and cut search. *Operations Research*, 21:123–134, 1973.

[42] F. Glover. Polyhedral convexity cuts and negative edge extensions. *Zeitschrift für Operations Research*, 18:181–186, 1974.

[43] F. Glover, G. Kochenberger, and Y. Du. Quantum bridge analytics i: a tutorial on formulating and using qubo models. *4OR*, 17(4):335–371, 2019.

[44] O. Günlük and Y. Pochet. Mixing mixed-integer inequalities. *Mathematical Programming*, 90:429–457, 2001. doi:10.1007/PL00011430.

[45] P. Hansen, B. Jaumard, M. Ruiz, and J. Xiong. Global minimization of indefinite quadratic functions subject to box constraints. *Naval Research Logistics (NRL)*, 40(3):373–392, 1993. doi:10.1002/1520-6750(199304)40:3<373::AID-NAV3220400307>3.0.CO;2-A.

[46] D. Henrich. Initialization of parallel branch-and-bound algorithms. In *Second International Workshop on Parallel Processing for Artificial Intelligence(PPAI-93)*, August 1993.

[47] C. Hojny. Packing, partitioning, and covering symresacks. *Discrete Applied Mathematics*, 283:689–717, 2020. doi:10.1016/j.dam.2020.03.002.

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

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

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

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

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

[53] V. Kaibel, M. Peinhardt, and M. E. Pfetsch. Orbitopal fixing. *Discrete Optimization*, 8 (4):595–610, 2011. doi:10.1016/j.disopt.2011.07.001.

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

[55] J. E. Kelley. The cutting-plane method for solving convex programs. *Journal of the Society for Industrial and Applied Mathematics*, 8(4):703–712, 1960. doi:10.1137/0108053.

[56] A. Khajavirad. Packing circles in a square: a theoretical comparison of various convexification techniques. Technical report, Optimization Online, 2017. http://www.optimization-online.org/DB_HTML/2017/03/5911.html.

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

[58] S. Küçükyavuz. On mixing sets arising in chance-constrained programming. *Mathematical Programming*, 132:31–56, 2012. doi:10.1007/s10107-010-0385-3.

[59] H. W. Kuhn and A. W. Tucker. Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability, 1950*, pages 481–492, Berkeley and Los Angeles, 1951. University of California Press.

[60] B. Legat, O. Dowson, J. Garcia, and M. Lubin. MathOptInterface: a data structure for mathematical optimization problems. *INFORMS Journal on Computing*, in press.

[61] L. Liberti. Reformulations in mathematical programming: automatic symmetry detection and exploitation. *Mathematical Programming*, 131(1):273–304, Feb 2012. doi:10.1007/s10107-010-0351-0.

[62] L. Liberti and J. Ostrowski. Stabilizer-based symmetry breaking constraints for mathematical programs. *Journal of Global Optimization*, 60:183–194, 2014.

[63] LINDO. Api users manual. `http://www.lindo.com/`, 2003.

[64] I. Ljubic. Solving Steiner Trees — Recent Advances, Challenges and Perspectives. *Networks*, 2020. Accepted for publication.

[65] J. Luedtke, S. Ahmed, and G. L. Nemhauser. An integer programming approach for linear programs with probabilistic constraints. *Mathematical Programming*, 122:247–272, 2010. doi:10.1007/s10107-008-0247-4.

[66] A. Mahajan and T. Munson. Exploiting second-order cone structure for global optimization. Technical Report ANL/MCS-P1801-1010, Argonne National Laboratory, 2010. URL `http://www.optimization-online.org/DB_HTML/2010/10/2780.html`.

[67] S. Maher, M. Miltenberger, J. P. Pedroso, D. Rehfeldt, R. Schwarz, and F. Serrano. PySCIPOpt: Mathematical programming in python with the SCIP optimization suite. In *Mathematical Software – ICMS 2016*, pages 301–307. Springer International Publishing, 2016. doi:10.1007/978-3-319-42432-3_37.

[68] S. J. Maher, T. Fischer, T. Gally, G. Gamrath, A. Gleixner, R. L. Gottwald, G. Hendel, T. Koch, M. E. Lübbecke, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, S. Schenker, R. Schwarz, F. Serrano, Y. Shinano, D. Weninger, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 4.0. Technical report, Optimization Online, 2017. URL `http://www.optimization-online.org/DB_HTML/2017/03/5895.html`.

[69] C. D. Maranas and C. A. Floudas. Finding all solutions of nonlinearly constrained systems of equations. *Journal of Global Optimization*, 7(2):143–182, 1995. doi:10.1007/BF01097059.

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

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

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

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

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

[75] F. Matter and M. E. Pfetsch. Presolving for mixed-integer semidefinite optimization. Technical report, Optimization Online, 2021. `http://www.optimization-online.org/DB{_}HTML/2021/10/8614.html`.

[76] G. P. McCormick. Computability of global solutions to factorable nonconvex programs: Part I – convex underestimating problems. *Mathematical Programming*, 10(1):147–175, 1976. doi:10.1007/BF01580665.

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

[78] B. Müller, F. Serrano, and A. Gleixner. Using Two-Dimensional Projections for Stronger Separation and Propagation of Bilinear Terms. *SIAM Journal on Optimization*, 30(2): 1339–1365, 2020. doi:10.1137/19m1249825.

[79] L.-M. Munguía, G. Oxberry, and D. Rajan. Pips-sbb: A parallel distributed-memory branch-and-bound algorithm for stochastic mixed-integer programs. In *2016 IEEE Inter-*

national Parallel and Distributed Processing Symposium Workshops (IPDPSW), pages 730–739, 2016. doi:10.1109/IPDPSW.2016.159.

[80] L.-M. Munguía, G. Oxberry, D. Rajan, and Y. Shinano. Parallel pips-sbb: multi-level parallelism for stochastic mixed-integer programs. *Computational Optimization and Applications*, 73(2):575–601, 2019. doi:10.1007/s10589-019-00074-0. URL `https://doi.org/10.1007/s10589-019-00074-0`.

[81] G. Muñoz and F. Serrano. Maximal quadratic-free sets. *Integer Programming and Combinatorial Optimization Proceedings*, 21:307–321, 2020. doi:10.1007/978-3-030-45771-6_24.

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

[83] D. Pecin, A. Pessoa, M. Poggi, and E. Uchoa. Improved branch-cut-and-price for capacitated vehicle routing. In J. Lee and J. Vygen, editors, *Integer Programming and Combinatorial Optimization*, pages 393–403, Cham, 2014. Springer International Publishing.

[84] T. Polzin. *Algorithms for the Steiner problem in networks*. PhD thesis, Saarland University, 2003.

[85] T. Polzin and S. V. Daneshmand. *Extending Reduction Techniques for the Steiner Tree Problem*, pages 795–807. Springer, Berlin, Heidelberg, 2002. doi:10.1007/3-540-45749-6_69.

[86] A. Qualizza, P. Belotti, and F. Margot. Linear programming relaxations of quadratically constrained quadratic programs. In J. Lee and S. Leyffer, editors, *Mixed Integer Nonlinear Programming*, pages 407–426, New York, NY, 2012. Springer New York. doi:10.1007/978-1-4614-1927-3_14.

[87] T. Ralphs, Y. Shinano, T. Berthold, and T. Koch. Parallel solvers for mixed integer linear optimization. In Y. Hamadi and L. Sais, editors, *Handbook of parallel constraint reasoning*, pages 283–336. Springer, 2018. doi:10.1007/978-3-319-63516-3_8.

[88] D. Rehfeldt and T. Koch. Implications, conflicts, and reductions for Steiner trees. ZIB-Report 20-28, Zuse Institute Berlin, 2020.

[89] D. Rehfeldt and T. Koch. On the exact solution of prize-collecting Steiner tree problems. *INFORMS Journal on Computing*, 2021. Accepted for publication.

[90] D. Rehfeldt and T. Koch. Implications, conflicts, and reductions for steiner trees. In M. Singh and D. P. Williamson, editors, *Proceedings: Integer Programming and Combinatorial Optimization – 22nd International Conference, IPCO 2021*, volume 12707 of *LNCS*, pages 473–487. Springer, 2021. doi:10.1007/978-3-030-73879-2_33.

[91] D. Rehfeldt, H. Franz, and T. Koch. Optimal connected subgraphs: Formulations and algorithms. ZIB-Report 20-23, Zuse Institute Berlin, 2020.

[92] S. Røpke. Branching decisions in branch-and-cut-and-price algorithms for vehicle routing problems, 2012. Presentation in Column Generation.

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

[94] D. Salvagnin. A dominance procedure for integer programming. Master's thesis, Universtà degli studi di Padova, 2005.

[95] D. Salvagnin. Symmetry breaking inequalities from the Schreier-Sims table. In W.-J. van Hoeve, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 521–529, Cham, 2018. Springer.

[96] M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6(4):445–454, 1994. doi:10.1287/ijoc.6.4.445.

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

[98] M. Schneider, N. Gama, P. Baumann, and L. Nobach. SVP challenge (2010). *URL: http://latticechallenge.org/svp-challenge*.

[99] F. Serrano, R. Schwarz, and A. Gleixner. On the relation between the extended supporting hyperplane algorithm and Kelley's cutting plane algorithm. *Journal of Global Optimization*, 78(1):161–179, 2020. doi:10.1007/s10898-020-00906-y.

[100] H. D. Sherali and C. H. Tuncbilek. A global optimization algorithm for polynomial programming problems using a reformulation-linearization technique. *Journal of Global Optimization*, 2(1):101–112, 1992.

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

[102] 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, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-24025-6.

[103] 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, 2016. doi:10.1109/IPDPS.2016.56.

[104] Y. Shinano, T. Berthold, and S. Heinz. Paraxpress: an experimental extension of the fico xpress-optimizer to solve hard mips on supercomputers. *Optimization Methods and Software*, 33(3):530–539, 2018. doi:10.1080/10556788.2018.1428602. URL `https://doi.org/10.1080/10556788.2018.1428602`.

[105] Y. Shinano, S. Heinz, S. Vigerske, and M. Winkler. Fiberscip—a shared memory parallelization of scip. *INFORMS Journal on Computing*, 30(1):11–30, 2018. doi:10.1287/ijoc.2017.0762. URL `https://doi.org/10.1287/ijoc.2017.0762`.

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

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

[108] N. Tateiwa, Y. Shinano, S. Nakamura, A. Yoshida, S. Kaji, M. Yasuda, and K. Fujisawa. Massive parallelization for finding shortest lattice vectors based on ubiquity generator framework. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020. doi:10.1109/SC41405.2020.00064.

[109] N. Tateiwa, Y. Shinano, K. Yamamura, A. Yoshida, S. Kaji, M. Yasuda, and K. Fujisawa. Cmap-lap: Configurable massively parallel solver for lattice problems. Technical Report 21-16, ZIB, Takustr. 7, 14195 Berlin, 2021.

[110] M. Tawarmalani and N. V. Sahinidis. A polyhedral branch-and-cut approach to global optimization. *Mathematical Programming*, 103(2):225–249, 2005. doi:10.1007/s10107-005-0581-8.

[111] H. Tuy. Concave programming with linear constraints. *Doklady Akademii Nauk*, 159: 32–35, 1964.

[112] S. Vahdati Daneshmand. *Algorithmic approaches to the Steiner problem in networks*. PhD thesis, Universität Mannheim, 2004.

[113] F. Vanderbeck. Branching in branch-and-price: A generic scheme. *Mathematical Programming*, 130(2):249–294, 2011.

[114] J. P. Vielma, I. Dunning, J. Huchette, and M. Lubin. Extended formulations in mixed integer conic quadratic programming. *Mathematical Programming Computation*, 9(3): 369–418, 2016. doi:10.1007/s12532-016-0113-y.

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

[116] D. Villeneuve, J. Desrosiers, M. Lübbecke, and F. Soumis. On compact formulations for integer programs solved by column generation. *Annals of Operations Research*, 139(1): 375–388, 2005. doi:10.1007/s10479-005-3455-9.

[117] F. Wegscheider. Exploiting symmetry in mixed-integer nonlinear programming. Master's thesis, Zuse Institute Berlin, 2019.

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

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

[120] J. M. Zamora and I. E. Grossmann. Continuous Global Optimization of Structured Process Systems Models. *Comp. Chem. Eng.*, 22(12):1749–1770, 1998.

[121] M. Zhao, K. Huang, and B. Zeng. A polyhedral study on chance constrained program with random right-hand side. *Mathematical Programming*, 166:19–64, 2017. doi:10.1007/s10107-016-1103-6.

Author Affiliations

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

Mathieu Besançon
Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany
E-mail: besancon@zib.de
ORCID: 0000-0002-6284-3033

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

Antonia Chmiela
Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany
E-mail: chmiela@zib.de
ORCID: 0000-0002-4809-2958

Tim Donkiewicz
RWTH Aachen University, Lehrstuhl für Operations Research, Kackertstr. 7, 52072 Aachen, Germany
E-mail: tim.donkiewicz@rwth-aachen.de
ORCID: 0000-0002-5721-3563

Jasper van Doornmalen
Technische Universiteit Eindhoven, Department of Mathematics and Computer Science, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
E-mail: m.j.v.doornmalen@tue.nl
ORCID: 0000-0002-2494-0705

Leon Eifler
Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany
E-mail: eifler@zib.de
ORCID: 0000-0003-0245-9344

Oliver Gaul
RWTH Aachen University, Lehrstuhl für Operations Research, Kackertstr. 7, 52072 Aachen, Germany
E-mail: oliver.gaul@rwth-aachen.de
ORCID: 0000-0002-2131-1911

Gerald Gamrath
Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany
and I^{2}DAMO GmbH, Englerallee 19, 14195 Berlin, Germany

E-mail: gamrath@zib.de
ORCID: 0000-0001-6141-5937

Ambros Gleixner
Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany
E-mail: gleixner@zib.de
ORCID: 0000-0003-0391-5903

Leona Gottwald
Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany
E-mail: gottwald@zib.de
ORCID: 0000-0002-8894-5011

Christoph Graczyk
Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany
E-mail: graczyk@zib.de

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

Alexander Hoen
Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany
E-mail: hoen@zib.de
ORCID: 0000-0003-1065-1651

Christopher Hojny
Technische Universiteit Eindhoven, Department of Mathematics and Computer Science, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
E-mail: c.hojny@tue.nl
ORCID: 0000-0002-5324-8996

Rolf van der Hulst
University of Twente, Department of Discrete Mathematics and Mathematical Programming, P.O. Box 217, 7500 AE Enschede, The Netherlands
E-mail: r.p.vanderhulst@utwente.nl

Thorsten Koch
Technische Universität Berlin, Chair of Software and Algorithms for Discrete Optimization, Straße des 17. Juni 135, 10623 Berlin, Germany, and
Zuse Institute Berlin, Department A^{2}IM, Takustr. 7, 14195 Berlin, Germany
E-mail: koch@zib.de
ORCID: 0000-0002-1967-0077

Marco Lübbecke
RWTH Aachen University, Lehrstuhl für Operations Research, Kackertstr. 7, 52072 Aachen, Germany
E-mail: marco.luebbecke@rwth-aachen.de
ORCID: 0000-0002-2635-0522

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

Frederic Matter
Technische Universität Darmstadt, Fachbereich Mathematik, Dolivostr. 15, 64293 Darmstadt, Germany
E-mail: matter@mathematik.tu-darmstadt.de
ORCID: 0000-0002-0499-1820

Erik Mühmer
RWTH Aachen University, Lehrstuhl für Operations Research, Kackertstr. 7, 52072 Aachen
E-mail: erik.muehmer@rwth-aachen.de
ORCID: 0000-0003-1114-3800

Benjamin Müller
Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany
E-mail: benjamin.mueller@zib.de
ORCID: 0000-0002-4463-2873

Marc E. Pfetsch
Technische Universität Darmstadt, Fachbereich Mathematik, Dolivostr. 15, 64293 Darmstadt, Germany
E-mail: pfetsch@mathematik.tu-darmstadt.de
ORCID: 0000-0002-0947-7193

Daniel Rehfeldt
Zuse Institute Berlin, Department A^{2}IM, Takustr. 7, 14195 Berlin, Germany
E-mail: rehfeldt@zib.de
ORCID: 0000-0002-2877-074X

Steffan Schlein
RWTH Aachen University, Lehrstuhl für Operations Research, Kackertstr. 7, 52072 Aachen, Germany
E-mail: steffan.schlein@rwth-aachen.de

Franziska Schlösser
Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany
E-mail: schloesser@zib.de

Felipe Serrano
Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany
E-mail: serrano@zib.de
ORCID: 0000-0002-7892-3951

Yuji Shinano
Zuse Institute Berlin, Department A^{2}IM, Takustr. 7, 14195 Berlin, Germany
E-mail: shinano@zib.de
ORCID: 0000-0002-2902-882X

Boro Sofranac
Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany and
Technische Universität Berlin, Straße des 17. Juni 135, 10623 Berlin, Germany
E-mail: sofranac@zib.de
ORCID: 0000-0003-2252-9469

Mark Turner
Zuse Institute Berlin, Department A^{2}IM, Takustr. 7, 14195 Berlin, Germany
and Chair of Software and Algorithms for Discrete Optimization, Institute of Mathematics, Technische Universität Berlin, Straße des 17. Juni 135, 10623 Berlin, Germany
E-mail: turner@zib.de
ORCID: 0000-0001-7270-1496

Stefan Vigerske
GAMS Software GmbH, c/o Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany
E-mail: svigerske@gams.com

Fabian Wegscheider
Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany
E-mail: wegscheider@zib.de

Philipp Wellner
E-mail: p.we@fu-berlin.de

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

Jakob Witzig
Zuse Institute Berlin, Department AIS^{2}T, Takustr. 7, 14195 Berlin, Germany
E-mail: witzig@zib.de
ORCID: 0000-0003-2698-0767

Appendices

A Detailed Computational Results to Section 4.14 (Performance Impact of Updates for Nonlinear Constraints)

The following table lists the results for running the classic code (SCIP 7) and the new code (SCIP 8) for each considered instance of MINLPLib. Only results for the non-permuted instances are given.

Column "time/gap" gives the time it took to solve the instance to optimality (with respect to specified gap tolerances) or the gap at termination if solving stopped at the time limit. If the time or gap of a version is not worse than the other version, the time or gap is printed in a bold font. If a version did not return a result or the reported bounds conflict with best known bounds, then "fail" is printed.

For the classic version, columns "quad", "soc", "abspow", and "nonlin" give the number of quadratic, second-order cone, abspower, and nonlinear constraints, respectively, after presolve. Due to the reformulations that are applied in presolve, nonlinear constraints are sums of convex or concave functions, quadratic terms (including bilinear products) are parts of quadratic constraint, unless an upgrade to a soc constraint was possible. Monomials of odd degree, signpowers, and monomials of even degree with fixed sign are handled by the abspower constraint handler.

For the new version, columns "quad", "bilin", "soc", "convex", "concave", "quot", "persp", "def" give the number of expressions for which the detection algorithm of the nonlinear handlers quadratic, bilinear, soc, convex, concave, quotient, perspective, and default, respectively (see Sections 4.3–4.8 and 4.2.6) reported success, that is, registered themselve for domain propagation or separation after presolve. Recall that by default the quadratic nonlinear handler only gets active for propagable quadratic expressions (see Section 4.3.1) and the convex and concave nonlinear handlers only handle nontrivial expressions (Section 4.6.1). Further, the nonlinear handler for bilinear expressions (Section 4.5) currently registers itself for any product of two non-binary variables (original or auxiliary) and only checks when called later whether linear inequalities in the two variables are available, because the latter are computed after the extended formulations are initialized. Columns "minor" and "rlt" report the number of cuts that were generated by the respective separators (Sections 4.10 and 4.9) and got added to the LP. Here, if cuts were generated but not applied, a zero is printed.

The last row summarizes on how many instances each constraint handler, nonlinear handler, or separator was used, i.e., the number of nonzeros in each column.

	classic					new									
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
4stufen	61.4%	26			16	**14.6%**	11	26	5	5	12		153			
abel	0.8s	1				**0.0s**							58			
alan	**0.1s**	1				**0.1s**	1		1				4			
alkyl	**0.2s**	9				0.3s	3	8					32			
alkylation	1.2%	7		1		**0.13%**	5	7	1			3	28			
arki0001	**0.1s**	1				0.3s							1026			
arki0002	∞	1			1824	∞			1824	912			6258			
arki0003	7.5s	4669		360		**2.1s**	1080	77					6243			
arki0004	∞	3336			54	∞	1040	4114			2027		8289			
arki0005	∞	2376				∞		19					4414			
arki0006	∞	2376				∞	1	19	1				3405			
arki0008	fail					∞	1750	3549	1003	1002	1500		10913		121116	0
arki0009	∞	40		50	191	**50.9%**	90		201	109			3669			
arki0010	∞	20		25	96	**45.2%**	45		106	59			1809			
arki0011	∞			135	286	∞	135		295	159			7125			
arki0012	∞			135	286	∞	135		295	159			7125			
arki0013	∞			135	286	∞	135		295	159			7260			
arki0014	∞			135	286	∞	135		295	159			7125			
arki0015	**3%**	231			992	**3.1%**		472	824	614			4904			
arki0016	∞	1983			2652	∞	1440	4629	12	2652			16634			0
arki0017	>1000%	1828			2201	**>1000%**	727	4015	26	2200			14310			0
arki0018	∞	1			9804	∞		9804		9804			39218			
arki0019	149%	1018			1	**45.2%**			1225	509	509		4489			
arki0020	147%	2522			1	**31.8%**			3136	1261	1261		11319			
arki0021	>1000%	6372			1	**27.3%**			7056	3186	3186		26859			
arki0022	139%	8302			1	**22.4%**			14400	4151	4151		45407			
arki0023	∞	17770			1	**18.8%**			27225	8885	8885		89993			
arki0024	∞	3422		32	47	**7.4%**	90	663	55	55			3715			
autocorr_bern20-03	0.0s					**0.0s**										
autocorr_bern20-05	**12.6s**					**14.2s**										
autocorr_bern20-10	**113.4s**					**134.8s**										

	classic					new									
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
autocorr_bern20-15	**401.1s**					**407.4s**										
autocorr_bern25-03	0.0s					**0.0s**										
autocorr_bern25-06	225.9s					**150.7s**										
autocorr_bern25-13	3411.4s					**1787.5s**										
autocorr_bern25-19	121%					**59.2%**										
autocorr_bern25-25	358%					**228%**										
autocorr_bern30-04	**35.7s**					**41.3s**										
autocorr_bern30-08	37.7%					**18.2%**										
autocorr_bern30-15	103%					**73.1%**										
autocorr_bern30-23	491%					**240%**										
autocorr_bern30-30	666%					**403%**										
autocorr_bern35-04	**70.7s**					90.3s										
autocorr_bern35-09	133%					**51.8%**										
autocorr_bern35-18	**340%**					**278%**										
autocorr_bern35-26	>1000%					**692%**										
autocorr_bern35-35	>1000%	12752				**942%**										
autocorr_bern35-35fix	>1000%					**942%**										
autocorr_bern40-05	**2235.3s**					**2702.2s**										
autocorr_bern40-10	353%					**146%**										
autocorr_bern40-20	672%					**521%**										
autocorr_bern40-30	>1000%					**789%**										
autocorr_bern40-40	>1000%					**>1000%**										
autocorr_bern45-05	40.6%					**10.2%**										
autocorr_bern45-11	380%					**269%**										
autocorr_bern45-23	>1000%					**767%**										
autocorr_bern45-34	>1000%					**971%**										
autocorr_bern45-45	>1000%					**>1000%**										
autocorr_bern50-06	128%					**74.3%**										
autocorr_bern50-13	**603%**					**502%**										
autocorr_bern50-25	>1000%					**893%**										
autocorr_bern50-38	**>1000%**					**>1000%**										
autocorr_bern50-50	**>1000%**					**>1000%**										
autocorr_bern55-06	173%					**90.2%**										
autocorr_bern55-14	>1000%					**660%**										
autocorr_bern55-28	>1000%					**>1000%**										
autocorr_bern55-41	**>1000%**					**>1000%**										
autocorr_bern55-55	**>1000%**					**>1000%**										
autocorr_bern60-08	470%					**367%**										
autocorr_bern60-15	**>1000%**					**881%**										
autocorr_bern60-30	**>1000%**					**>1000%**										
autocorr_bern60-45	**>1000%**					**>1000%**										
autocorr_bern60-60	**>1000%**					**>1000%**										
ball_mk2_10	**0.0s**	1				**0.0s**	1						21			
ball_mk2_30	**0.0s**	1				0.0s	1						61			
ball_mk3_10	**0.0s**	1				**0.0s**	1						1			
ball_mk3_20	**0.0s**	1				**0.0s**	1						41			
ball_mk3_30	**0.0s**	1				**0.0s**	1						61			
ball_mk4_05	**1.9s**	1				**1.6s**	1			1			10			
ball_mk4_10	**1456.0s**	1				1836.8s	1			1			20			
ball_mk4_15	∞	1				∞	1			1			30			
batch	0.6s				2	**0.3s**				11			46			
batch0812	0.8s				2	**0.4s**				20			80			
batch0812_nc	**1.9s**	89			11	**1.6s**	5	6	4			4	58			
batch_nc	**0.9s**	36			6	1.1s		38				5	74			
batchdes	**0.1s**				2	**0.1s**				5			23			
batchs101006m	8.3s				2	**5.6s**				29			110			
batchs121208m	15.8s				2	**6.5s**				35			130			
batchs151208m	19.8s				2	**8.4s**				38			136			
batchs201210m	**22.8s**				2	**20.4s**				43			156			
bayes2_10	∞	52				∞	54	384					502			0
bayes2_20	∞	54				∞	55	385					505			0
bayes2_30	∞	54				∞	55	385					505			0
bayes2_50	**1404.7s**	54				**1680.7s**	55	385	1				505			0
bchoco05	**5%**	53		3	13	∞		14					133	14	13	
bchoco06	**4.4%**	101		11		∞		19					207	33		
bchoco07	**4.5%**	261		25		∞		26					309	64	11	
bchoco08	**1.4s**	652		48		fail		33					516	149	137	
bearing	**9.5s**	8	1	2	3	fail	2	8		1	1		43			
beuster	>1000%	44			16	**408%**	18	30		4	4	12	209			2
blend029	15.2s	12				**3.0s**	8	32					83			
blend146	3.2%	24				**630.8s**	16	128					260			
blend480	3013.6s	32				**148.8s**	24	188					364			
blend531	194.4s	30				**20.0s**	22	156					304			
blend718	107%	24				**1475.0s**	16	120					247			
blend721	51.7s	24				**20.7s**	16	128					264			
blend852	0.08%	32				**71.7s**	24	188					371			
btest14	22.5%	41		5	45	**0.98%**	43	71					372		48727	
camcns	∞	101		9	66	∞	4	159		30	12	36	647			
camshape100	**7.3%**	101				**6.1%**	100	197					397			3772
camshape200	**14.8%**	201				**13.3%**	200	397					797			3011
camshape400	**18.6%**	401				**18.4%**	400	797					1597			325
camshape800	**21.4%**	801				**21.6%**	800	1597					3197			228
cardqp_inlp	7.6%					**2893.5s**										
cardqp_iqp	7.3%					**2900.3s**										
carton7	379.4s	60				**18.7s**	12	6					116			1
carton9	193%	68				**41.2s**	12	21					172			79
casctanks	**177.4s**	263		5	59	∞	35	233					977			2

	classic					new										
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
case_1scv2	**>1000%**	999		300		∞	696	609			4	9	4410			
catmix100	∞	200				∞	200	200					701			
catmix200	∞	400				∞	400	400					1401			
catmix400	∞	800				∞	800	800					2801			
catmix800	∞	1600				∞	1600	1600					5601			
cecil_13	1247.5s	174			174	**640.5s**				58			725	12		
celar6-sub0	∞					**3281.3s**										
cesam2cent	fail					**>1000%**	28	28	1	13	157		629			
cesam2log	∞	28			170	∞	29	185	1	13	157		786			
chain100	∞	102			101	∞		99		101	101		605			
chain200	∞	202			201	∞		199		201	201		1205			
chain400	∞	402			401	∞		399		401	401		2405			
chain50	∞	52			51	∞		49		51	51		305			
chakra	**0.0s**				22	0.1s							118			
chance	0.1s	1			1	**0.0s**			1				11			
chem	1650.5s	11			10	**1012.9s**	1	20				10	44			
chenery	4.6s	11			20	**0.5s**	1	16		8	8		108			
chimera_k64ising-01	**10.4s**					44.6s										
chimera_k64ising-02	**10.0s**					49.3s										
chimera_k64maxcut-01	**181.8s**					604.6s										
chimera_k64maxcut-02	**152.7s**					509.1s										
chimera_lga-01	**3.2%**					**2.9%**										
chimera_lga-02	**4.4%**					**3.8%**										
chimera_mgw-c16-2031-01	**22.3%**					32.3%										
chimera_mgw-c16-2031-02	**22.4%**					**27.9%**										
chimera_mgw-c8-439-onc8-001	**141.3s**					**132.4s**										
chimera_mgw-c8-439-onc8-002	**224.0s**					**194.4s**										
chimera_mgw-c8-507-onc8-01	**5.6%**					**4.9%**										
chimera_mgw-c8-507-onc8-02	**4.2%**					**4%**										
chimera_mis-01	10.2s					**4.4s**										
chimera_mis-02	10.3s					**6.0s**										
chimera_rfr-01	**8.2%**					25.2%										
chimera_rfr-02	**8.1%**					17.7%										
chimera_selby-c16-01	**42.2%**					**44.7%**										
chimera_selby-c16-02	**30.0%**					**29.3%**										
chimera_selby-c8-onc8-01	8.7%					**6.9%**										
chimera_selby-c8-onc8-02	**3.3%**					**3.8%**										
chp_partload	∞	516	6	82	383	∞	363	1058	7	31	16	181	2633		2866	10
chp_shorttermplan1a	**13.3s**	444		130		**15.3s**	86						230	44		33
chp_shorttermplan1b	2%	836		310		**1.2%**	768						2500	288		
chp_shorttermplan2a	32.0s	432		670		**12.4s**	768						2165	216		0
chp_shorttermplan2b	15.0%	1152		192		**1398.6s**	672	192					2592	192	418	2
chp_shorttermplan2c	∞	864		1148		∞	1344						3660	384		0
chp_shorttermplan2d	∞	1968		576		∞	1344	240					5720	480	1239	
circle	0.0s		10			**0.0s**	10		10				4			
clay0203h	**1.6s**	168				49.7s	24	24		9	9	12	117	6		
clay0203hfsg	**1.4s**	288				**1.3s**	24	24		9	9	12	117	6		
clay0203m	0.6s	24				**0.4s**	24						39			
clay0204h	**4.9s**	224				6.4s	32	32		12	12	16	156	8		
clay0204hfsg	8.6s	384				**3.7s**	32	32		12	12	16	156	8		
clay0204m	**1.1s**	32				**1.4s**	32						52			
clay0205h	**38.7s**	280				50.7s	40	40		15	15	20	195	10		
clay0205hfsg	51.6s	480				**21.9s**	40	40		15	15	20	195	10		
clay0205m	**6.9s**	40				9.1s	40						65			
clay0303h	**2.4s**	252				13.1s	36	36		9	9	18	180	18		
clay0303hfsg	**2.6s**	432				**2.4s**	36	36		9	9	18	180	18		
clay0303m	**0.6s**	36				**0.6s**	36						55			
clay0304h	**7.3s**	336				183.2s	48	48		12	12	24	240	24		
clay0304hfsg	**6.8s**	576				10.6s	48	48		12	12	24	240	24		
clay0304m	1.4s	48				**1.1s**	48						74			
clay0305h	**72.7s**	420				**61.9s**	60	60		15	15	30	300	30		
clay0305hfsg	**156.3s**	720				fail	60	60		15	15	30	300	30		
clay0305m	**9.0s**	60				**9.6s**	58						89			
color_lab2_4x0	∞					∞										
color_lab3_3x0	32.7%					**20.4%**										
color_lab3_4x0	255%					**138%**										
color_lab6b_4x20	∞					∞										
cont6-qq	**20.9%**	39602				∞	39602	39601					237410			
contvar	106%	258		4	194	**67.2%**	35	71		187	68	25	867			
crossdock_15x7	98.9%					**64.0%**										
crossdock_15x8	210%					**124%**										
crudeoil_lee1_05	**1.4s**	6				2.7s	2	42					119	2		
crudeoil_lee1_06	**5.8s**	30				**4.9s**	2	60					170	2		
crudeoil_lee1_07	**3.6s**	36				**2.9s**	2	72					204	2		
crudeoil_lee1_08	22.4s	42				**6.5s**	2	84					238	2		
crudeoil_lee1_09	**12.9s**	48				20.7s	2	84					238	2		
crudeoil_lee1_10	32.7s	54				**11.7s**	2	108					306	2		
crudeoil_lee2_05	17.7s	16				**13.5s**	2	27					74			
crudeoil_lee2_06	29.6s	64				**18.3s**	2	134					348	5		
crudeoil_lee2_07	161.2s	70				**27.6s**	2	162					421	5		
crudeoil_lee2_08	**40.7s**	95				57.9s	4	184					478	2		0
crudeoil_lee2_09	**47.4s**	109				**50.2s**	2	218					567	5		
crudeoil_lee2_10	**67.7s**	123				110.7s	2	246					640	5		
crudeoil_lee3_05	**29.6s**	106				**34.7s**	5	199					523	1		
crudeoil_lee3_06	**64.6s**	141				**54.5s**	1	275					717	3		
crudeoil_lee3_07	**91.1s**	176				259.1s	3	333					863	1		
crudeoil_lee3_08	384.1s	211				**233.9s**	1	415					1073	3		

	classic					new									
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
crudeoil_lee3_09	**273.5s**	246				**267.5s**	1	485					1251	3		
crudeoil_lee3_10	**363.9s**	281				594.0s	1	555					1429	3		
crudeoil_lee4_05	15.4s	58				**8.2s**	1	146					366	4		
crudeoil_lee4_06	**15.2s**	92				**17.3s**	1	184					461	4		
crudeoil_lee4_07	**60.8s**	103				**52.9s**	1	206					522	4		
crudeoil_lee4_08	**27.5s**	130				43.5s	1	260					651	4		
crudeoil_lee4_09	129.9s	141				**69.0s**	1	298					746	4		
crudeoil_lee4_10	**64.2s**	168				**62.5s**	1	336					841	4		
crudeoil_li01	0.65%	56				**0.47%**		56					228			
crudeoil_li02	**1.1%**	15				**1.3%**		15					58			
crudeoil_li03	**2%**	192				**2.1%**		192					720			
crudeoil_li05	**10.7%**	192				**10.3%**		192					716			
crudeoil_li06	**552.0s**	192				**531.0s**		192					719			
crudeoil_li11	**1.7%**	192				7.1%		192					720			
crudeoil_li21	**2.2%**	192				5.9%		192					720			
crudeoil_pooling_ct1	37.7%	37				**25.1%**	3	76					233			10
crudeoil_pooling_ct2	0.14%	70				**14.6s**	2	70					280			22
crudeoil_pooling_ct3	49.6%	182				**35.1%**	2	274					839			19
crudeoil_pooling_ct4	3.6%	95				**12.8s**	1	95					365			21
crudeoil_pooling_dt1	fail	570				**6.6%**	2	570					2470			86
crudeoil_pooling_dt2	**19.0%**	1106				**16.5%**	2	1106					4424			134
crudeoil_pooling_dt3	**20.4%**	2707				∞	1	2707					10795			0
crudeoil_pooling_dt4	**8.2%**	1121				**8.3%**	1	1121					4307			59
csched1	**4.3s**	4			3	**3.9s**	1	7		3	3	3	25			
csched1a	**0.4s**	5			3	7.6s	1	9		3		9	31			
csched2	**775%**	29			28	>1000%	1	57		28	28	28	200			
csched2a	>1000%	30			28	**>1000%**	1	84		28		84	256			
cvxnonsep_normcon20	1.4s	1			1	**0.1s**							42			
cvxnonsep_normcon20r	0.2s			20		**0.1s**							80			
cvxnonsep_normcon30	5.3s	1			1	**0.1s**							62			
cvxnonsep_normcon30r	0.4s			30		**0.1s**							120			
cvxnonsep_normcon40	64.5s	1			1	**0.1s**							82			
cvxnonsep_normcon40r	0.4s			40		**0.2s**							160			
cvxnonsep_nsig20	**106.4s**				1	253.0s				1			42			
cvxnonsep_nsig20r	0.1s				20	**0.1s**							80			
cvxnonsep_nsig30	**3.1%**				1	18.2%				1			62			
cvxnonsep_nsig30r	0.3s				30	**0.1s**							120			
cvxnonsep_nsig40	**17.6%**				1	25.5%				1			82			
cvxnonsep_nsig40r	0.3s				38	**0.1s**				2			162			
cvxnonsep_pcon20	**0.2s**	1			19	4.3s				190			400			
cvxnonsep_pcon20r	0.3s				19	**0.2s**				19			96			
cvxnonsep_pcon30	**0.4s**	1			29	77.7s				435			900			
cvxnonsep_pcon30r	1.2s				29	**0.2s**				29			146			
cvxnonsep_pcon40	**0.6s**	1			39	350.5s				705			1448			
cvxnonsep_pcon40r	**0.5s**				39	0.7s				32			173			
cvxnonsep_psig20	**17.2s**				1	**18.9s**				1			43			
cvxnonsep_psig20r	**0.2s**				21	**0.2s**							83			
cvxnonsep_psig30	2638.6s				1	**116.1s**				1			63			
cvxnonsep_psig30r	**0.1s**				31	0.2s							123			
cvxnonsep_psig40	**9%**				1	**8.3%**				1			83			
cvxnonsep_psig40r	0.3s				41	**0.2s**							164			
demo7	0.0s	2				**0.0s**	2						19			
densitymod	∞				106	∞				105			422			
dispatch	0.0s	2				**0.0s**	1	3		1			16		6	
dtoc5	fail					∞	49998						249995			
du-opt	**15.6s**	1				**17.2s**	1			1			21			
du-opt5	**0.5s**	1				**0.4s**	1			1			19			
edgecross10-010	0.1s					**0.1s**										
edgecross10-020	0.8s					**0.3s**										
edgecross10-030	**0.4s**					1.3s										
edgecross10-040	**5.3s**					10.5s										
edgecross10-050	**16.2s**					29.8s										
edgecross10-060	569.0s	1				**38.4s**	1	27					1074			953
edgecross10-070	**72.9s**					174.8s										
edgecross10-080	194.3s	1				**53.0s**	1						1050			37
edgecross10-090	**1.2s**					3.4s										
edgecross14-019	**1.0s**					**0.9s**										
edgecross14-039	3230.2s	1				**26.4s**	1	155					784			253
edgecross14-058	**86.9s**					124.8s										
edgecross14-078	74.3%	1				**14.0%**	1						2364			167
edgecross14-098	**3200.7s**					32.0%										
edgecross14-117	**20.4%**					**22.9%**										
edgecross14-137	**1038.0s**					15.4%										
edgecross14-156	26.5%	1				**12.6%**	1						3563			263
edgecross14-176	**128.4s**					244.2s										
edgecross20-040	**14.3s**					33.0s										
edgecross20-080	**1714.8s**					2186.0s										
edgecross22-048	**30.3s**					64.3s										
edgecross22-096	**87.7%**					162%										
edgecross24-057	**403.1s**					1035.5s										
edgecross24-115	**141%**					199%										
eg_all_s	∞	24541		17	16878	**166%**	2619			1320			4055			
eg_disc2_s	∞	29815		21	16878	∞	2619			1320			4057			
eg_disc_s	∞	29715		22	16878	**154%**	2619			1320			4057			
eg_int_s	∞	24685		9	16878	**21.4%**	2619			1320			4055			
eigena2	∞	1276				∞		61250					67527		14447	
elec100	211%	5050			1	**145%**	4950	14850		4950			20502		8655	

	classic					new									
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
elec200	∞	20100			1	∞	19900	59700		19900			81002		13816	
elec25	165%	325			1	**61.2%**	300	900		300			1377		417914	427
elec50	193%	1275			1	**83.5%**	1225	3675		1225			5252		262433	76
elf	0.8s	27				**0.6s**	24	3	3				87			
emfl050_3_3	1.8s		522			**1.3s**	441		522				1449			
emfl050_5_5	fail		1850			**6.0s**	1225		1850				6825			
emfl100_3_3	**2.2s**		972			0.01%	891		972				2349			
emfl100_5_5	76.2s		3100			**26.3s**	2475		3100				9325			
eniplac	**1.0s**	21		24		2.5s	24						159	23		1
enpro48pb	5.2s				2	**1.9s**				13			52			
enpro56pb	**4.2s**				2	**3.5s**				12			51			
eq6_1	**293%**	29			28	546%	29	56		29			117		24483	
etamac	>1000%				19	**0.81%**				18	10		117			
ethanolh	131%	36		1	55	**0.2s**		3					39			
ethanolm	46.9%	24		1	31	**1.6s**		3					39			
ex1221	**0.0s**			2		**0.0s**							8	2		
ex1222	**0.0s**	1			1	**0.0s**					1		7			
ex1223	0.1s	1		2	1	**0.1s**	1			1			18			
ex1223a	**0.0s**	2		2		**0.0s**							14			
ex1223b	0.1s	1		2	1	**0.0s**	1			1			18			
ex1224	0.1s	2			3	**0.0s**				3	2		23			
ex1225	0.0s	1		2		**0.0s**		1					6			
ex1226	**0.0s**	1		1	2	**0.0s**	1	1					8			
ex1233	13.7%	33			12	**1.9%**	5	11		4			72	6		
ex1243	**0.8s**	12				1.1s	1	12		12	6	12	58	6		
ex1244	1076.2s	17				**26.2s**	1	16		16	6	16	82	5		
ex1252	**12.7s**	21		9		0.43%	3	15	3				68	24	9174	
ex1252a	18.3%	21		9		**0.2%**	3	15	3				68	24	10211	
ex1263	**0.2s**	4				0.3s		16					40	13		2
ex1263a	**0.4s**	4				**0.5s**		16					40	13		1
ex1264	**0.1s**	4				0.1s		16					40	12		0
ex1264a	0.3s	4				**0.1s**		16					40	12		1
ex1265	**0.1s**	5				0.2s		25					60	25		0
ex1265a	0.7s	5				**0.3s**		25					60	25		1
ex1266	**0.1s**	6				0.2s		36					84	24		0
ex1266a	**0.1s**	6				0.6s		36					84	24		7
ex14_1_1	0.6s	4		2		**0.0s**	4	1					12			
ex14_1_2	0.5s	15		1		**0.0s**	9	4					22			
ex14_1_3	0.1s	2			2	**0.0s**		1		2	2		12			
ex14_1_5	0.7s	8				**0.0s**							9			
ex14_1_6	0.0s	14				**0.0s**	6	2					31			
ex14_1_7	551.9s	42			8	**8.7s**	8	7		8	8		67			
ex14_1_8	0.2s	6			2	**0.1s**	4	3		5	5	2	20			
ex14_1_9	>1000%	2			2	**0.0s**	2	1		2	2		10			
ex14_2_1	**0.0s**	18			6	**0.0s**		3		9	3	3	29			
ex14_2_2	0.0s	8			4	**0.0s**	4	1		6	6	2	17			
ex14_2_3	0.0s	32			8	**0.0s**		4		12	4	4	38			
ex14_2_4	0.0s	48			6	**0.0s**	9	12		12	3	9	50			
ex14_2_5	0.0s	16		2	4	**0.0s**	6	4		9	7	6	24			
ex14_2_6	0.0s	24			6	**0.0s**	6	6		13	3	6	39			
ex14_2_7	0.1s	40			8	**0.0s**	8	8		16	4	8	49			
ex14_2_8	0.0s	16			4	**0.0s**	4	3		10	10	3	22			
ex14_2_9	0.0s	12			4	**0.0s**	4	3		10	10	3	22			
ex2_1_1	**0.1s**	1				**0.1s**							12			
ex2_1_10	0.1s	1				**0.1s**							32			
ex2_1_2	**0.0s**	1				**0.0s**							12			
ex2_1_3	**0.0s**	1				**0.0s**							10			
ex2_1_4	**0.0s**			1		**0.0s**							4			
ex2_1_5	0.0s	1				**0.0s**							16			
ex2_1_6	**0.1s**	1				**0.1s**							22			
ex2_1_7	**0.8s**	1				1.6s							42			
ex2_1_8	**0.1s**	1				**0.1s**							50			
ex2_1_9	**0.4s**	1				4.2s	1	22					34			
ex3_1_1	1.8s	3				**0.2s**	2	2	1				8			
ex3_1_2	0.0s	7				**0.0s**	2	5					14			
ex3_1_3	**0.1s**	3				**0.1s**	2						18			
ex3_1_4	0.1s	1				**0.1s**	1	3					10			0
ex3pb	**0.1s**				5	0.1s				2	2		19			
ex4	fail	26				**1.6s**	16						46			
ex4_1_1	0.2s			2	1	**0.1s**	1						8			
ex4_1_2	**0.3s**				1	0.4s	1						52			
ex4_1_3	0.2s				1	**0.2s**	1						7			
ex4_1_4	0.5s			1	1	**0.2s**							6			
ex4_1_5	∞	1			1	**1.7s**	1			1			5			
ex4_1_6	0.3s				1	**0.2s**							6			
ex4_1_7	**0.1s**			1	1	**0.1s**	1						6			
ex4_1_8	**0.0s**			2		**0.0s**							7			
ex4_1_9	**0.2s**				2	**0.1s**	1						7			
ex5_2_2_case1	0.7s	3				**0.1s**	1	2					14			
ex5_2_2_case2	0.3s	3				**0.1s**	1	2					14			
ex5_2_2_case3	0.2s	3				**0.0s**	1	2					14			
ex5_2_4	0.5s	4				**0.3s**	4	6					18			1
ex5_2_5	116%	12				**27.5%**	12	60					105			13
ex5_3_2	0.2s	9				**0.2s**	6	9					33			0
ex5_3_3	77.1%	35				**12.4%**	6	69					158			26
ex5_4_2	0.4s	3				**0.1s**	2	2	1				8			
ex5_4_3	**0.4s**	6			1	**0.3s**	4	8		2			25			0

	classic					new									
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
ex5_4_4	27.4s	9			1	**5.5s**	6	15		3			44			16
ex6_1_1	**188.7s**	5			6	>1000%	5	14		6	2		34			
ex6_1_2	**0.3s**	3			2	2.8s	3	3	1	2	1		12			
ex6_1_3	**1766.1s**	7			8	216%	7	27		8	3		54			
ex6_1_4	29.0s	4			3	**0.2s**	4	9					23			1
ex6_2_10	274%	19			28	**65.5%**	10	42		22	3	27	109			
ex6_2_11	fail	10			14	**>1000%**	1	21		6		9	51			
ex6_2_12	**0.11%**	11			16	1.7%	6	20		15	2	15	65			
ex6_2_13	369%	15			22	**66.1%**	8	28		19	3	21	87			
ex6_2_14	0.04%	13			20	**83.1s**	7	24		20	2	18	80			
ex6_2_5	>1000%	4			21	**884%**	1	24		11		3	70			
ex6_2_6	3.2%	1			9	**21.7s**	1	9		5			29			
ex6_2_7	>1000%	1			24	**>1000%**	1	27		15			80			
ex6_2_8	0.01%	1			8	**12.9s**	1	9		5			28			
ex6_2_9	562%	13			20	**94.3%**	7	28		19	2	18	82			
ex7_2_1	3.8%	27		2	7	**2.4%**	5	9			1	8	41			
ex7_2_2	**0.3s**	4			1	**0.3s**	4	4					17			
ex7_2_3	**236%**	13			2	**212%**	3	10		1	1	12	29			
ex7_2_4	27.1s	8		1	4	**6.7s**	2	8		2		4	34			
ex7_3_1	0.1s	8		4		**0.0s**	1	11					23		0	
ex7_3_2	0.3s	2			2	**0.0s**	1	2					8			
ex7_3_3	**0.1s**	2				**0.1s**	2	2					9			
ex7_3_4	8.5s	20		2		**1.2s**	2	9					36		15	
ex7_3_5	fail	28		4		**338.2s**	5	12					41		4862	
ex7_3_6	0.1s	1		2	1	**0.0s**							17			
ex8_1_3	∞	7				∞	1	28					46		8	
ex8_1_4	∞	1			1	**239.5s**	1			1			5			
ex8_1_5	∞	1			1	∞	1	1					10			
ex8_1_6	**0.1s**	3			1	0.1s	3						12			
ex8_1_7	**0.9s**	6		2	1	**0.7s**	1	1		3			18			
ex8_2_1b	**0.8s**	25			3	1.0s		50		5	2		146			
ex8_2_2b	2.4%	1875			13	**2682.8s**		6156		18	12		13923			
ex8_2_3b	**324.3s**	3125			6	0.15%		9065		11	5		19988			
ex8_2_4b	**0.5s**	75			7	0.9s		106		7	4		242			
ex8_2_5b	0.08%	3750			25	**1774.7s**		12312		28	22		17558			
ex8_3_1	**>1000%**	59		5	30	**>1000%**	20	204		15	15		413			94
ex8_3_11	>1000%	59		5	30	**>1000%**	20	204		15	15		413			32
ex8_3_12	∞	49			20	**906%**	20	204		15	15		418			22
ex8_3_13	416%	64		5	30	**139%**	20	184		10	10		398			45
ex8_3_14	∞	69			40	**>1000%**	20	174		20	10		398			12
ex8_3_2	**>1000%**	44		5		**>1000%**	20	189					338			380
ex8_3_3	>1000%	44		5		**>1000%**	20	189					338			2382
ex8_3_4	**179%**	44		5		**179%**	20	189					338			273
ex8_3_5	>1000%	49				**>1000%**	20	189					338			1065
ex8_3_7	fail					**>1000%**	25	241				1	430			4873
ex8_3_8	**207%**	55		10		**207%**	25	239					424			4854
ex8_3_9	**>1000%**	27				**>1000%**	10	92					192			37
ex8_4_1	141.5s	11				**0.8s**		10					64			
ex8_4_2	>1000%	32		19		**0.96%**		30					96			
ex8_4_3	∞	1			25	**6.7s**				25			179			
ex8_4_4	57.7s	13			6	**1.9s**		12		6	6		67			
ex8_4_5	**7.3s**	23				826.6s		11		3		11	83			
ex8_4_6	**0.68%**	17			24	∞		24		24	24		112			
ex8_4_7	35.0%	51			20	**13.5%**	20	40		10		10	224			
ex8_4_8_bnd	∞	104		7	40	∞	10	5		40	30		250		187	
ex8_5_1	∞	6		2	4	**3.4s**	2	8		1		2	30		7	
ex8_5_2	∞	6		2	4	**0.6%**	2	8		1		2	30		5	
ex8_5_3	fail					**0.5s**	2	5		3		1	27		2	
ex8_5_4	fail					**1.2s**	2	5		4		1	26		5	
ex8_5_5	fail					**1.1s**	3	7		3		2	30		1	
ex8_5_6	∞	9		3	5	**41.7s**	3	10		3		2	38		131	
ex8_6_1	fail					∞	36	85		44			404		1628	
ex8_6_2	**41.1%**	45			45	>1000%	72	85		81	45		412		162393	10
ex9_1_1	**0.0s**					0.0s	1	5					15			
ex9_1_2	**0.0s**					**0.0s**		4					12			
ex9_1_4	**0.0s**					**0.0s**		3					9			
ex9_1_5	**0.0s**					**0.0s**		5					15			
ex9_1_8	**0.0s**					**0.0s**	1	2					7			
ex9_2_2	0.1s	1				**0.0s**	1	1					9			
ex9_2_3	**0.0s**					0.2s	2	6					18			
ex9_2_4	0.1s	1				**0.0s**		2					10			
ex9_2_5	0.1s	1				**0.1s**		3					15			
ex9_2_6	0.1s	1				**0.0s**	3	4					20			
ex9_2_7	**0.1s**	1				**0.1s**		4					17			
ex9_2_8	**0.0s**					**0.0s**										
fac1	**0.1s**					0.3s										
fac2	**0.5s**				1	0.7s				3			62	3		
fac3	**0.2s**	1				0.9s	1			1			55			
faclay20h	**1042.4s**					1510.0s										
faclay25	**92.0%**					**98.1%**										
faclay30	**237%**					**205%**										
faclay30h	216%					**133%**										
faclay33	320%					**189%**										
faclay35	380%					**275%**										
faclay60	>1000%					**>1000%**										
faclay70	∞					**>1000%**										
faclay75	∞					∞										

	classic					new									
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
faclay80	∞					∞										
fdesign10	**0.0s**		1			**0.0s**	1		1				4			
fdesign25	**0.1s**		1			**0.0s**	1		1				4			
fdesign50	**0.1s**		1			**0.1s**	1		1				4			
feedtray	∞	134		18	72	**2.9s**	38	323		144	126	54	654			3
feedtray2	**0.1s**	147				0.1s	89	79	12				274			
filter	0.0s	3			3	**0.0s**	1	2		2	2	3	11			
fin2bb	**8.8s**	42				60.1s		42				21	189			
flay02h	0.6s				2	**0.4s**							8			
flay02m	0.4s				2	**0.3s**							8			
flay03h	**1.1s**				3	**1.3s**							12			
flay03m	**0.7s**				3	**0.7s**							12			
flay04h	22.0s				4	**7.3s**							16			
flay04m	**4.0s**				4	**3.9s**							16			
flay05h	**323.0s**				5	**356.1s**							20			
flay05m	**110.9s**				5	**131.0s**							20			
flay06h	**7.9%**				6	12.4%							24			
flay06m	**3.8%**				6	**3.8%**							24			
flowchan100fix	**0.4s**	400				1.1s		800					3200			
flowchan200fix	**0.9s**	800				fail										
flowchan400fix	**1.0s**	1600				fail										
flowchan50fix	**0.2s**	200				**0.2s**		400					1600			
fo7	**139.4s**				14	**160.6s**							42			
fo7_2	191.9s				14	**53.1s**							42			
fo7_ar25_1	**38.8s**				14	**44.5s**							42			
fo7_ar2_1	**47.4s**				14	**38.7s**							42			
fo7_ar3_1	**52.9s**				14	169.3s							42			
fo7_ar4_1	**54.6s**				14	**65.9s**							42			
fo7_ar5_1	**40.2s**				14	**42.9s**							42			
fo8	**179.9s**				16	**171.7s**							48			
fo8_ar25_1	**201.5s**				16	**240.0s**							48			
fo8_ar2_1	**404.5s**				16	**374.2s**							48			
fo8_ar3_1	**73.8s**				16	145.6s							48			
fo8_ar4_1	**52.8s**				16	144.5s							48			
fo8_ar5_1	**109.0s**				16	151.8s							48			
fo9	1082.0s				18	**530.3s**							54			
fo9_ar25_1	**19.2%**				18	**20.5%**							54			
fo9_ar2_1	**1577.5s**				18	2247.3s							54			
fo9_ar3_1	**149.6s**				18	**149.8s**							54			
fo9_ar4_1	218.1s				18	**119.2s**							54			
fo9_ar5_1	**331.5s**				18	1793.3s							54			
forest	43.1%	24				**85.1s**	23	90	1				205			11
fuel	**0.0s**	1		3		**0.0s**	4						18	3		
gabriel01	2300.2s	48				**263.4s**	32	256					436			
gabriel02	25.8%	96				**2372.0s**	64	512					782			
gabriel04	4.9%	128				**148.3s**	96	752					1120			
gabriel05	∞	192				∞	168	1284					1948			
gabriel06	∞	640				∞	608	4592					6578			
gabriel07	∞	800				∞	760	5740					8052			
gabriel08	∞	1600				∞	1520	20600					25864			
gabriel09	∞	288				∞	216	4284					5624			
gabriel10	∞	3200				∞	3040	77680					90768			
gams01	>1000%	110			111	**>1000%**	110			120			384			
gams02	**363%**	1			192	990%	193	192					1070	24		
gams03	**>1000%**	1				**>1000%**	1	50960					55042	3360		22
gancns	∞	187		19	165	∞	43	142		37	18	23	665			150
gasnet	148%	61		13	23	**75.2%**	19	30		3	3		240	11		
gasnet_al1	**0.73%**	256		10	35	**0.59%**	45	24		75	73	13	848	71		2
gasnet_al2	1.2%	256		10	35	**0.55%**	45	24		75	73	13	848	71		9
gasnet_al3	2%	256		10	35	**0.36%**	45	24		75	73	13	848	71		1
gasnet_al4	**0.4%**	256		10	35	1.4%	45	24		75	73	13	848	71		3
gasnet_al5	**0.77%**	256		10	35	2.3%	45	24		75	73	13	848	71		0
gasoil100	∞	2001				∞	401	1600		1			4604			
gasoil200	∞	4001				∞	801	3200		1			9004			
gasoil400	∞	8001				∞	1601	6400		1			17804			
gasoil50	∞	1001				∞	201	800		1			2404			
gasprod_sarawak01	2.6s	34				**0.6s**	17	34					123			3
gasprod_sarawak16	0.95%	544				**0.39%**	272	544					1968			55
gasprod_sarawak81	**0.41%**	2754				0.93%	1377	2754					9963			61
gastrans	**0.1s**	23		11		0.1s							111	1		
gastrans040	40.4s	183		39	51	**0.3s**	72	119	13	38	38	50	456			
gastrans135	∞	749		144	239	**40.7s**	316	582	21	164	164	239	2088			
gastrans582_cold13	∞	1012		186	288	**56.4s**	227	290	116	93	93	123	1729	50		
gastrans582_cold13_95	**3.0s**	1012		185	288	30.9s	209	267	129	85	85	113	1667	52		0
gastrans582_cold17	fail	1072		205	288	**7.6s**	247	305	111	109	109	126	1784	56		
gastrans582_cold17_95	∞	1072		205	288	**15.1s**	246	305	111	109	109	126	1784	56		
gastrans582_cool12	∞	1048		204	288	**8.7s**	235	303	120	106	106	126	1779	55		
gastrans582_cool12_95	∞	1048		201	288	**11.3s**	236	303	120	106	106	126	1779	55		
gastrans582_cool14	∞	1050		201	288	**9.8s**	235	299	119	104	104	125	1771	53		
gastrans582_cool14_95	∞	1050		201	288	∞	207	252	132	90	90	103	1651	61		
gastrans582_freezing27	54.4s	1094		205	261	**8.5s**	241	298	115	111	111	121	1790	47		
gastrans582_freezing27_95	56.1s	1098		209	262	**6.9s**	187	194	128	95	95	74	1542	50		0
gastrans582_freezing30	∞	1100		209	262	**81.6s**	175	184	128	91	91	66	1520	47		
gastrans582_freezing30_95	∞	1100		210	288	**25.6s**	200	230	128	95	95	89	1626	47		1
gastrans582_mild10	fail	1023		186	288	**8.9s**	226	297	118	95	95	126	1744	50		
gastrans582_mild10_95	fail	1023		187	288	∞	226	297	118	95	95	126	1744	50		
gastrans582_mild11	18.9s	1011		185	280	**7.6s**	228	290	122	98	98	122	1730	53		

	classic					new										
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
gastrans582_mild11_95	∞	1011		184	280	**21.8s**	198	243	135	91	90	101	1606	55		
gastrans582_warm15	∞	1010		183	288	**298.9s**	229	296	114	92	92	126	1746	47		
gastrans582_warm15_95	∞	1010		183	288	**7.4s**	229	296	114	92	92	126	1746	47		
gastrans582_warm31	∞	1004		181	288	**12.0s**	188	242	130	83	83	100	1610	50		
gastrans582_warm31_95	∞	1004		181	288	**8.0s**	231	303	117	91	91	130	1760	42		
gastransnlp	0.0s			22		**0.0s**							88			
gbd	**0.0s**	1				**0.0s**							4			
gear	**0.1s**	4				16.2s							14			
gear2	**0.1s**	4				18.0s							14			
gear3	**0.1s**	4				15.8s							14			
gear4	fail	3				**0.7s**		1					11			
genpooling_lee1	7.4s	20				**2.8s**	20	24					84			
genpooling_lee2	98.8s	30				**5.5s**	30	36					110			
genpooling_meyer04	121%	15				**42.9%**	15	66					144			
genpooling_meyer10	105%	33				**62.4%**	33	435					675			
genpooling_meyer15	102%	48				**52.8%**	48	990					1420			
ghg_1veh	**13.2s**	47		3	11	17.1s	9	18		6	4	2	104		17	
ghg_2veh	**5.9%**	119		6	23	**5%**	34	38	1	12	10	3	261		11	6
ghg_3veh	53.0%	210		9	36	**29.1%**	39	58		22	16	5	453		5852	3
gilbert	**0.1s**	2				1.0s							2003			
gkocis	0.0s				2	**0.0s**				2	2		10	2		
glider100	∞	1501		200	400	∞	600	706	99	200	101	4	4516			
glider200	∞	3001		400	800	∞	1200	1406	199	400	201	4	9016			
glider400	∞	6000		801	1600	∞	2400	2806	399	800	401	4	18016			
glider50	∞	751		100	200	∞	300	356	49	100	51	4	2266			
graphpart_2g-0044-1601	0.1s					**0.1s**										
graphpart_2g-0055-0062	**0.3s**					1.2s										
graphpart_2g-0066-0066	**0.9s**					5.8s										
graphpart_2g-0077-0077	**1.3s**					5.2s										
graphpart_2g-0088-0088	**1.7s**					3.2s										
graphpart_2g-0099-9211	**4.4s**					11.1s										
graphpart_2g-1010-0824	**1.7s**					10.3s										
graphpart_2pm-0044-0044	**0.2s**					**0.1s**										
graphpart_2pm-0055-0055	**0.3s**					1.4s										
graphpart_2pm-0066-0066	**0.8s**					1.3s										
graphpart_2pm-0077-0777	**1.9s**					2.5s										
graphpart_2pm-0088-0888	**1.4s**					4.7s										
graphpart_2pm-0099-0999	**4.0s**					8.2s										
graphpart_3g-0234-0234	**0.6s**					3.1s										
graphpart_3g-0244-0244	**0.8s**					1.6s										
graphpart_3g-0333-0333	**0.9s**					1.5s										
graphpart_3g-0334-0334	**2.8s**					**3.0s**										
graphpart_3g-0344-0344	**2.3s**					4.3s										
graphpart_3g-0444-0444	**12.7s**					18.1s										
graphpart_3pm-0234-0234	**0.7s**					3.3s										
graphpart_3pm-0244-0244	**1.2s**					1.7s										
graphpart_3pm-0333-0333	**1.2s**					**1.0s**										
graphpart_3pm-0334-0334	**3.5s**					**2.8s**										
graphpart_3pm-0344-0344	**12.1s**					**14.9s**										
graphpart_3pm-0444-0444	**51.6s**					97.0s										
graphpart_clique-20	**2.4s**					4.4s										
graphpart_clique-30	**16.5s**					**15.8s**										
graphpart_clique-40	**126.5s**					**113.3s**										
graphpart_clique-50	1294.6s					**673.3s**										
graphpart_clique-60	87.2%					**3503.6s**										
graphpart_clique-70	149%					**76.1%**										
gsg_0001	93.9s	1			20	**38.4s**	1	20					94			
gtm	**0.0s**				1	**0.0s**				6			41			
hadamard_4	**0.2s**					1.6s										
hadamard_5	36.4s					**22.4s**										
hadamard_6	**478%**					**476%**										
hadamard_7	>1000%					**>1000%**										
hadamard_8	∞					∞										
hadamard_9	fail					∞										
harker	0.2s				1	**0.0s**	1						42			
haverly	0.6s	3				**0.1s**	3	2					12			
hda	213%	155		3	62	**10.7%**	17	135		78	75	8	783	15		0
heatexch_gen1	**0%**	56			12	88.4%	40	60		24	24	40	256			
heatexch_gen2	**0.76%**	54			17	18.6%	37	77		32	48	52	335	8		0
heatexch_gen3	**228.3s**	290			61	141%	160	470		120	180	220	1657	26		0
heatexch_spec1	18.8%	29			12	**80.4s**	5	11		18	2	7	94	6		
heatexch_spec2	46.5%	42			17	**20.1s**	17	16		26	20	15	164	12		
heatexch_spec3	>1000%	170			61	**0.91%**	10	60		110	55	50	659	35		
heatexch_trigen	**712.9s**	60			24	1.5%	9	18		27	21	20	182			
hhfair	fail					fail										
himmel11	**0.0s**	4				**0.0s**	3	7					22			
himmel16	**4.6s**	19				**5.3s**	10	7		10			37		2621	
hmittelman	**0.0s**					**0.0s**										
house	0.5s	3				**0.3s**	1	3	1				12			
hs62	**0.01%**	4			1	**0.01%**	3	9		4	2	7	26		521	
hvb11	53.5s	17				**27.7s**		16					82			
hybriddynamic_fixed	**0.0s**					**0.0s**										
hybriddynamic_fixedcc	fail	1				fail	1	18					69			
hybriddynamic_var	**1.2s**	15		8		2.3s	3	9					49	1		
hybriddynamic_varcc	fail	39		2		fail	1	58					169			
hydro	fail					**0.0s**	6						38			
hydroenergy1	**0.03%**	46				0.04%	46	69					253			

	classic					new									
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
hydroenergy2	**1.2%**	92				**1.5%**	92	161					529			
hydroenergy3	**2.2%**	161				**2.1%**	161	299					943			
ibs2	1.1%				10	**35.5s**				3000			9010	1500		
immun	**0.0s**	1				0.1s							12			
infeas1	**53.1%**	1319			135	**46.9%**				104	40		4059			
ising2_5-300_5555	**17.0%**					21.5%										
jbearing100	∞	1				fail										
jbearing25	∞	1				∞	1			1			1251			
jbearing50	∞	1				fail										
jbearing75	∞	1				fail										
jit1	0.5s				1	**0.1s**							26			
johnall	900.4s	3991				**10.5s**							776			
junkturn	fail					∞		299985					699970			
kall_circles_c6a	**533.9s**	22				**450.9s**	21	43					96			1131
kall_circles_c6b	408.4s	22				**236.6s**	21	43					96			743
kall_circles_c6c	**213%**	29				355%	28	57					121			81
kall_circles_c7a	691.9s	29				**256.2s**	28	57					121			661
kall_circles_c8a	43.7%	37				**3304.2s**	36	73					149			580
kall_circlespolygons_c1p11	0.0s	21				**0.0s**		19					80			
kall_circlespolygons_c1p12	∞	21				**253.2s**		19					80			
kall_circlespolygons_c1p13	∞	21				**62.2s**		19					80			
kall_circlespolygons_c1p5a	**788%**	106				∞		101					360			
kall_circlespolygons_c1p5b	∞	631				∞		611					2022			
kall_circlespolygons_c1p6a	∞	904				∞		877					2878			
kall_circlesrectangles_c1r11	0.0s	23				**0.0s**		21					88			
kall_circlesrectangles_c1r12	∞	23				**60.6s**		21					88		182	
kall_circlesrectangles_c1r13	237.4s	23				**7.6s**		21					88		69	
kall_circlesrectangles_c6r1	**61.6%**	133				305%	15	141					455		52924	
kall_circlesrectangles_c6r29	**135%**	283				∞	15	283					939		6831	
kall_circlesrectangles_c6r39	**144%**	466				∞	15	457					1526		14677	
kall_congruentcircles_c31	0.4s	4				**0.3s**	3	7					26			0
kall_congruentcircles_c32	**0.3s**	4				**0.3s**	3	7					26			0
kall_congruentcircles_c41	**0.1s**	6				**0.1s**	6	6					20			0
kall_congruentcircles_c42	**0.3s**	7				0.4s	6	13					39			0
kall_congruentcircles_c51	7.5s	11				**5.8s**	10	21					55			147
kall_congruentcircles_c52	**1.9s**	11				**1.7s**	10	21					55			44
kall_congruentcircles_c61	106.6s	16				**45.8s**	15	31					74			567
kall_congruentcircles_c62	14.9s	16				**1.9s**	15	31					74			30
kall_congruentcircles_c63	5.6s	16				**4.0s**	15	31					74			38
kall_congruentcircles_c71	744.1s	22				**477.6s**	21	43					96			1873
kall_congruentcircles_c72	105.6s	22				**27.7s**	21	43					96			197
kall_diffcircles_10	151%	45				**2003.5s**	45	81					164		123771	45
kall_diffcircles_5a	37.6s	11				**2.2s**	10	21					55			
kall_diffcircles_5b	102.0s	11				**6.6s**	10	21					55			
kall_diffcircles_6	**4.8s**	16				6.1s	15	31					74			
kall_diffcircles_7	621.5s	22				**28.8s**	21	43					96			37
kall_diffcircles_8	123.2s	28				**89.2s**	28	49					107		6391	1
kall_diffcircles_9	**122.5s**	36				536.2s	36	64					134		17711	324
kall_ellipsoids_tc02b	**44.3%**	102		1	1	**44.0%**	3	48	6				232		15132	
kall_ellipsoids_tc03c	116%	209		1	3	**92.3%**	9	108	9				398		42873	
kall_ellipsoids_tc05a	∞	1227	45			∞	20	540	15				2396		18160	
kissing2	∞	10001				**>1000%**	4950	36884			1		44244		1300	
knp3-12	257%	78				**146%**	66	198					349		198594	1114
knp4-24	690%	300				**319%**	276	1104					1597		291309	3110
knp5-40	>1000%	820				**316%**	780	3900					5121		12435	134
knp5-41	>1000%	861				**325%**	820	4100					5372		8739	29
knp5-42	>1000%	903				**330%**	861	4305					5629		22458	224
knp5-43	>1000%	946				**342%**	903	4515					5892		8298	15
knp5-44	>1000%	990				**337%**	946	4730					6161		15500	110
korcns	2221.7s	29		1	20	**781.9s**	4	33		10	5	8	163			1
kport20	**422.6s**	57			1	820.1s							158			
kport40	**20.6%**	125			1	**21.7%**							355			
kriging_peaks-full010	**3.3s**	30			20	**2.9s**	10	20		30	30		144			
kriging_peaks-full020	**30.0s**	60			40	0.01%	20	40		60	60		284			
kriging_peaks-full030	**58.5s**	90			60	0.01%	30	60		90	90		424			
kriging_peaks-full050	**195.8s**	150			100	0.01%	50	100		150	150		704			
kriging_peaks-full100	**2425.7s**	300			200	0.01%	100	200		300	300		1404			
kriging_peaks-full200	74.9%	600			400	**0.01%**	200	400		600	600		2804			
kriging_peaks-full500	784%	1500			1000	**271%**	500	1000		1500	1500		7004			
kriging_peaks-red010	**39.4s**	11			20	0.01%	13	50	10	38	52		206			
kriging_peaks-red020	**0.01%**	21			40	**0.01%**	24	100	20	84	96		406			
kriging_peaks-red030	**0.01%**	31			60	**0.01%**	47	150	30	166	104		606			
kriging_peaks-red050	34.9%	51			100	**0.01%**	66	250	50	230	220		1006			
kriging_peaks-red100	318%	101			200	**0.02%**	134	500	100	488	412		2006			
kriging_peaks-red200	>1000%	201			400	**0.09%**	259	1000	200	924	876		4006			
kriging_peaks-red500	>1000%	501			1000	**419%**	650	2500	500	2348	2152		10006			
lakes	fail					∞	1	12		6	6		200			
launch	22.3s	25		6	25	**0.1s**	3	15					105			
least	**3220.2s**	7			6	∞	1	12		18	12		61			
like	∞	360			121	fail										
linear	2.1s				1	**0.0s**							42			
lip	**0.1s**				1	0.9s							50			
lop97ic	85.7%	40				**3.1%**	39						2896			
lop97icx	55.8%	40				**20.5s**	39						348			
lukvle10	∞	3998			1001	∞	1998	1000		1000			6998			
m3	0.3s				6	**0.2s**							18			
m6	**2.0s**				12	**2.4s**							36			

	classic					new									
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
m7	4.4s				14	**3.3s**							42			
m7_ar25_1	**1.7s**				14	3.4s							42			
m7_ar2_1	16.4s				14	**12.7s**							42			
m7_ar3_1	**12.9s**				14	**11.0s**							42			
m7_ar4_1	3.5s				14	**2.1s**							42			
m7_ar5_1	**8.7s**				14	13.7s							42			
mathopt1	**0.0s**	2				0.1s	2	2					10		10	0
mathopt2	**0.0s**	1			1	**0.0s**										
mathopt5_4	**0.0s**	1		1	1	2.0s	1						10			
mathopt5_7	**0.2s**				1	**0.2s**	1						7			
mathopt5_8	0.2s				1	**0.1s**	1						6			
maxcsp-ehi-85-297-12	∞					∞										
maxcsp-ehi-85-297-36	∞					∞										
maxcsp-ehi-85-297-71	∞					∞										
maxcsp-ehi-90-315-70	∞					∞										
maxcsp-geo50-20-d4-75-36	160.6s					**61.7s**										
maxcsp-langford-3-11	∞					∞										
maxmin	81.0%	78			78	**40.6%**	66	132		78			349		51364	2080
maxmineig2	fail	90				fail		294					574			
mbtd	220%					**133%**										
meanvar	**0.0s**	1				**0.0s**	1			1			8			
meanvar-orl400_05_e_7	4.3%	1	400			**0.7%**	1		400	1			2401			
meanvar-orl400_05_e_8	**87.5s**	1				246.7s	1			1			250	83		
meanvarx	**0.0s**	1				0.1s	1			1			8			
meanvarxsc	0.1s	1				**0.1s**	1			1			8			
methanol100	∞	4501				∞	601	2400		1			7098			
methanol200	∞	9001				∞	1201	4800		1			13998			
methanol400	∞	18001				∞	2401	9600		1			27798			
methanol50	∞	2251				∞	301	1200		1			3636			
mhw4d	∞	4		2	1	**20.1s**	1	1		3			18			
milinfract	**33.7s**	1				270%	1	500				1000	2004			
minlphi	fail					**0.7s**	3	4		4		4	24			
minlphix	fail					fail										
minsurf100	∞	10300			1	fail	10100		102	9998	200		15702			
minsurf25	**155%**	2650			1	fail	2600		102	2498	50		4002			
minsurf50	∞	5200			1	fail	5100		102	4998	100		7902			
minsurf75	∞	7750			1	**352%**	7600		102	7498	150		11802			
multiplants_mtg1a	**856.0s**	32		3		13.1%	25	34					175			
multiplants_mtg1b	**194%**	32		3		503%	22	33					168			
multiplants_mtg1c	**582%**	32		3		828%	25	34					202			
multiplants_mtg2	11.2%	42		4		**2.1%**	33	45					231			
multiplants_mtg5	**13.2%**	53		3		20.6%	43	51					207			
multiplants_mtg6	**15.4%**	70		4		31.9%	61	69					379			
multiplants_stg1	∞	67			30	∞	1	67					347			
multiplants_stg1a	∞	49			21	**>1000%**	1	49					313			
multiplants_stg1b	∞	55			24	**>1000%**	1	55					364			
multiplants_stg1c	fail					**>1000%**	1	43					338			
multiplants_stg5	∞	49			21	∞	1	49					302			
multiplants_stg6	∞	65			28	∞	1	65					481			
nd_netgen-2000-2-5-a-a-ns_7	**0%**		1999			128%			1999				11994			
nd_netgen-2000-3-4-b-a-ns_7	219.1s		1988			**59.4s**			1988				11928			
nd_netgen-3000-1-1-b-b-ns_7	612.9s		3000			**43.1s**			3000				18000			
ndcc12	∞	46				∞	44	528					1144			3
ndcc12persp	∞	46				∞	44	43	1				294			
ndcc13	**27.4%**	42				∞	42	481	5				1106			8
ndcc13persp	fail					**44.7%**	42	28	14				224			
ndcc14	∞	54				**81.7%**	54	756					1620			2
ndcc14persp	**81.2%**	54				∞	54	52	2				365			
ndcc15	**29.4%**	40				∞	36	540					1152			6
ndcc15persp	∞	40				∞	36	31	5				227			
ndcc16	∞	60				**85.4%**	60	960					2040			
ndcc16persp	∞	60				∞	60	60					420			
nemhaus	**0.0s**					**0.0s**										
netmod_dol1	**13.3%**	1				85.4%										
netmod_dol2	**40.9s**	1				**41.2s**							14			
netmod_kar1	15.0s	1				**6.1s**							10			
netmod_kar2	**5.6s**	1				**6.0s**							10			
ngone	>1000%	4951				**>1000%**	4852	195		4852			690		60898	
no7_ar25_1	**55.1s**				14	69.3s							42			
no7_ar2_1	115.6s				14	**26.1s**							42			
no7_ar3_1	**165.7s**				14	226.9s							42			
no7_ar4_1	434.5s				14	**325.0s**							42			
no7_ar5_1	**130.1s**				14	167.4s							42			
nous1	16.8%	29				**12.2s**	8	50					124			12
nous2	2.1s	29				**0.5s**	8	50					124			12
nuclear104	∞	3221				∞	1976	3130					37236			
nuclear10a	∞	3130				∞	1976	3130					29046			12
nuclear10b	**>1000%**	3026				∞	1976	3130					7206			
nuclear14	∞	602				∞	360	584					2844			
nuclear14a	>1000%	584				**998%**	360	584					2544			19
nuclear14b	**77.4%**	560				105%	360	584					1344			
nuclear25	∞	628				∞	375	608					3089			
nuclear25a	∞	608				**>1000%**	375	608					2699			48
nuclear25b	635%	583				**103%**	375	608					1399			
nuclear49	∞	1374				∞	833	1332					9715			
nuclear49a	**>1000%**	1332				∞	833	1332					7965			43
nuclear49b	∞	1283				**143%**	833	1332					3065			

	classic					new									
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
nuclearva	∞	267				∞	163	258					1002			
nuclearvb	fail					∞	163	258					1002			
nuclearvc	∞	267				fail										
nuclearvd	∞	267				∞	163	258					1002			
nuclearve	∞	267				∞	163	258					1002			
nuclearvf	∞	267				∞	163	258					1002			
nvs01	**0.1s**	5		1	2	**0.1s**	1	2		1	3	1	15			
nvs02	**0.0s**	4				0.0s	3	7					21			
nvs03	**0.0s**	1		1		**0.0s**							7			
nvs04	**0.0s**	1		1		0.1s	1	2					10		15	
nvs05	**3.3s**	15		3	7	**3.6s**	3	4		6	1	2	44		63	
nvs06	0.1s	3		2	3	**0.0s**	1	2					13			
nvs07	**0.0s**	2		1		**0.0s**		1					9			
nvs08	**0.1s**	1		1	2	**0.1s**							13			
nvs09	28.5%	9			21	**0.1s**					20		74			
nvs10	**0.0s**	3				**0.0s**	2			2			6			
nvs11	**0.1s**	4				**0.1s**	4			4			4			
nvs12	**0.1s**					**0.1s**										
nvs13	**0.1s**	6				0.3s	6	10		3			24		90	
nvs14	**0.0s**	4				0.0s	3	7					21			
nvs15	0.1s					**0.0s**	1	1					9		2	1
nvs16	**0.0s**	4		2		0.1s	1	8					17		0	
nvs17	**0.3s**	8				2.5s	8	21		4			40		510	
nvs18	**0.2s**	7				0.8s	7	15		4			31		150	
nvs19	**0.4s**	9				3.6s	9	28		5			49		664	
nvs20	**0.7s**	1		16		1.0s	1	120					186		114	
nvs21	0.1s	5		3		**0.0s**		2					12			
nvs22	fail	15		3	7	**0.1s**	4	6		7	1	2	53		2	
nvs23	**0.6s**	10				7.6s	10	36		4			61		1250	
nvs24	**0.9s**	11				14.5s	11	45		7			70		2154	
o7	**13.8%**				14	**16.9%**							42			
o7_2	2219.0s				14	**1468.6s**							42			
o7_ar25_1	**750.1s**				14	**649.2s**							42			
o7_ar2_1	266.2s				14	**202.6s**							42			
o7_ar3_1	**1051.6s**				14	1513.8s							42			
o7_ar4_1	**2404.8s**				14	**2381.0s**							42			
o7_ar5_1	3305.3s				14	**1162.3s**							42			
o8_ar4_1	**21.2%**				16	**19.0%**							48			
o9_ar4_1	**33.0%**				18	**28.8%**							54			
oaer	0.1s				1	**0.0s**				2	2		10	1		
oil	53.8%	427		6	598	**17.0%**	71	217		32	32	13	1319	78		1
oil2	30.0s	278		4	536	**12.7s**	12	153		27	27	14	816	4		
optcdeg2	**76.3%**	1		49999		∞	49999						299996			
optmass	**305%**	5002				**305%**							25015			
ortez	**0.1s**	26			6	**0.1s**	1	13					68			2
orth_d3m6	∞	238		5		**247%**		30					177		8838	
orth_d3m6_pl	>1000%	177		1		**197%**		175					302		39621	711
orth_d4m6_pl	**138%**	171		1		**118%**		105					222		5646	406
otpop	**0.0s**	13			17	0.6s	1	16		1	1		115			
p_ball_10b_5p_2d_h	**17.4s**	300				40.8s	50	200		50	50	100	850	100		
p_ball_10b_5p_2d_m	**1.6s**	50				**1.8s**	50						116			
p_ball_10b_5p_3d_h	**47.1s**	400				102.4s	50	300		50	50	150	1150	150		
p_ball_10b_5p_3d_m	**4.6s**	50				**5.1s**	50						130			
p_ball_10b_5p_4d_h	**91.5s**	500				193.6s	50	400		50	50	200	1450	200		
p_ball_10b_5p_4d_m	8.4s	50				**6.1s**	50						132			
p_ball_10b_7p_3d_h	**552.9s**	560				1657.0s	70	420		70	70	210	1610	210		
p_ball_10b_7p_3d_m	**27.4s**	70				**28.6s**	70						172			
p_ball_15b_5p_2d_h	**72.5s**	450				152.3s	75	300		75	75	150	1275	150		
p_ball_15b_5p_2d_m	**3.6s**	75				5.5s	70						160			
p_ball_20b_5p_2d_h	**99.7s**	600				242.0s	100	400		100	100	200	1700	200		
p_ball_20b_5p_2d_m	**4.5s**	100				6.5s	100						220			
p_ball_20b_5p_3d_h	**351.6s**	800				1009.0s	100	600		100	100	300	2300	300		
p_ball_20b_5p_3d_m	**21.6s**	100				**26.9s**	100						230			
p_ball_30b_10p_2d_h	∞	1800				∞	300	1200		300	300	600	5100	600		
p_ball_30b_10p_2d_m	**175%**	300				∞	300						640			
p_ball_30b_5p_2d_h	**462.9s**	900				758.0s	150	600		150	150	300	2550	300		
p_ball_30b_5p_2d_m	**7.2s**	150				**7.5s**	150						320			
p_ball_30b_5p_3d_h	**610.1s**	1200				1880.8s	150	900		150	150	450	3450	450		
p_ball_30b_5p_3d_m	**22.0s**	150				40.4s	150						330			
p_ball_30b_7p_2d_h	**214%**	1260				∞	210	840		210	210	420	3570	420		
p_ball_30b_7p_2d_m	**140.5s**	210				386.5s	210						448			
p_ball_40b_5p_3d_h	**1705.1s**	1600				86.9%	200	1200		200	200	600	4600	600		
p_ball_40b_5p_3d_m	**47.2s**	200				78.8s	200						430			
p_ball_40b_5p_4d_h	**36.1%**	2000				∞	200	1600		200	200	800	5800	800		
p_ball_40b_5p_4d_m	**200.0s**	200				**242.8s**	200						440			
parabol5_2_1	∞	1				**3435.9s**							804			
parabol5_2_2	40.4%	1		200		**19.5%**	201						1800			
parabol5_2_3	**44.3%**	1		200		∞	201						1800			
parabol5_2_4	**>1000%**	40001				**>1000%**	40001	119600					200602		1339	
parabol_p	fail	193				∞	193						24386			
parallel	fail	109				**8.3s**	5	101				109	230			
pb302035	**319%**					**275%**										
pb302055	**301%**					**275%**										
pb302075	368%					**286%**										
pb302095	211%					**153%**										
pb351535	**250%**					**205%**										
pb351555	**209%**					**183%**										

	classic					new										
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
pb351575	**269%**					**281%**										
pb351595	**311%**					**295%**										
pedigree_ex1058	**234.3s**					772.9s										
pedigree_ex485	**27.4s**					103.5s										
pedigree_ex485_2	**10.7s**					37.4s										
pedigree_sim2000	34.4%					**1.4%**										
pedigree_sim400	**2.8%**					**2.6%**										
pedigree_sp_top4_250	**149.0s**					256.0s										
pedigree_sp_top4_300	**30.6s**					63.4s										
pedigree_sp_top4_350tr	**26.1s**					41.5s										
pedigree_sp_top5_200	**204.5s**					2921.6s										
pedigree_sp_top5_250	**82.2s**					1019.2s										
pindyck	**28.4%**	32			32	37.6%	16	47		33	30	15	252			
pinene100	∞	1501				∞	601	2700		1			6766			
pinene200	∞	3001				∞	1201	5400		1			13366			
pinene50	∞	751				fail										
pointpack02	**0.0s**	1				**0.0s**	1	2					12		0	0
pointpack04	**0.1s**	6				0.2s	6	12					35		4	0
pointpack06	4.8s	15				**3.4s**	15	30					70		2589	104
pointpack08	**68.0s**	28				**81.4s**	28	56					117		16362	1227
pointpack10	**15.2%**	45				**13.1%**	45	90					176		37684	9120
pointpack12	43.3%	66				**32.9%**	66	132					247		59064	11990
pointpack14	**28.2%**	91				**31.8%**	91	182					330		490522	8572
pollut	0.0s				1	**0.0s**				20			102			
pooling_adhya1pq	2.5s	20				**0.6s**	4	20					72			0
pooling_adhya1stp	9.2s	40				**2.5s**	4	40					125			19
pooling_adhya1tp	2.6s	20				**0.8s**		20					73			1
pooling_adhya2pq	1.5s	20				**0.7s**	4	20					72			0
pooling_adhya2stp	**2.9s**	40				**2.5s**	4	40					125			17
pooling_adhya2tp	0.8s	20				**0.6s**		20					73			0
pooling_adhya3pq	2.0s	32				**1.2s**	4	32					115			5
pooling_adhya3stp	9.2s	64				**2.5s**	4	64					199			13
pooling_adhya3tp	**1.5s**	32				**1.6s**		32					116			17
pooling_adhya4pq	**0.5s**	32				1.2s		40					122			11
pooling_adhya4stp	3.0s	64				**2.2s**		80					212			8
pooling_adhya4tp	2.6s	32				**1.7s**		40					122			23
pooling_bental4pq	**0.1s**	6				**0.1s**		6					23			0
pooling_bental4stp	0.1s	12				**0.1s**	3	12					39			0
pooling_bental4tp	**0.0s**	6				0.1s	3	6					22			
pooling_bental5pq	**0.3s**	60				1.1s		60					207			0
pooling_bental5stp	**4.2s**	120				8.6s		120					354			37
pooling_bental5tp	**0.1s**	60				0.2s		60					207			
pooling_digabel16	0.85%	81				**18.8s**	81	144					396			
pooling_digabel18	7.1%	390				**0.02%**	390	360					958			
pooling_digabel19	0.19%	128				**0.04%**	128	212					552			
pooling_epa1	fail	58		6	16	**23.6s**	38	45	2	24	8		220	2		2
pooling_epa2	∞	98		9	24	**1.8%**	68	114		36	12		442	9		5
pooling_epa3	∞	294		30	80	**2.9%**	201	420		120	40		1484	30		5
pooling_foulds2pq	0.1s	16				**0.0s**	8	16					58			
pooling_foulds2stp	**0.3s**	32				0.5s	8	32					102			0
pooling_foulds2tp	0.1s	16				**0.1s**		16					60			
pooling_foulds3pq	**0.1s**	512				0.7s		512					1696			
pooling_foulds3stp	∞	1024				**32.6s**		1024					2880			2
pooling_foulds3tp	**0.1s**	512				0.3s		512					1696			
pooling_foulds4pq	**0.3s**	512				0.6s		512					1696			
pooling_foulds4stp	2982.4s	1024				**22.1s**		1024					2880			1
pooling_foulds4tp	**0.1s**	512				0.3s		512					1696			
pooling_foulds5pq	**0.1s**	512				0.6s		512					1632			
pooling_foulds5stp	4.5%	1024				**120.0s**		1024					2752			9
pooling_foulds5tp	**0.1s**	512				0.3s		512					1632			
pooling_haverly1pq	0.0s	4				**0.0s**	2	4					15			
pooling_haverly1stp	**0.1s**	8				**0.1s**	4	8					26			
pooling_haverly1tp	**0.0s**	4				0.0s	2	4					15			
pooling_haverly2pq	**0.1s**	4				**0.1s**	2	4					15			
pooling_haverly2stp	0.2s	8				**0.1s**	4	8					26			
pooling_haverly2tp	0.1s	4				**0.0s**	2	4					15			
pooling_haverly3pq	**0.0s**	4				0.0s	2	4					15			
pooling_haverly3stp	**0.1s**	8				**0.1s**	4	8					26			
pooling_haverly3tp	**0.1s**	4				**0.1s**	2	4					15			
pooling_rt2pq	0.7s	18				**0.5s**		18					66			0
pooling_rt2stp	**0.8s**	36				1.1s		36					114			0
pooling_rt2tp	**0.3s**	18				0.4s		18					66			0
pooling_sppa0pq	1.7%	329				**1.2%**		329					1103			6122
pooling_sppa0stp	∞	658				**283%**		658					1877			2998
pooling_sppa0tp	1.6%	329				**0.67%**		329					1103			11943
pooling_sppa5pq	2.5%	968				**1.8%**		968					3100			1836
pooling_sppa5stp	∞	1936				**396%**		1936					5232			94
pooling_sppa5tp	**3.3%**	968				**4.1%**		968					3100			669
pooling_sppa9pq	**0.08%**	1828				**0.07%**		1828					5760			361
pooling_sppa9stp	∞	3656				**275%**		3820					9866			64
pooling_sppa9tp	46.0%	1828				**1.2%**		1992					5934			7
pooling_sppb0pq	∞	1153				**7.1%**		1153					3748			5268
pooling_sppb0stp	∞	2306				**>1000%**		2306					6343			374
pooling_sppb0tp	∞	1153				**7.4%**		1153					3748			1397
pooling_sppb2pq	∞	3093				**9.6%**		3093					9748			829
pooling_sppb2stp	∞	6186				**267%**		6186					16403			124
pooling_sppb2tp	**6%**	3093				12.0%		3093					9748			

	classic					new									
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
pooling_sppb5pq	∞	7947				∞		7947					24603			648
pooling_sppb5stp	∞	15894				**101%**		15894					41259			25
pooling_sppb5tp	**1.6%**	7947				66.2%		7947					24603			
pooling_sppc0pq	**13.3%**	2826				**15.2%**		2826					9074			1407
pooling_sppc0stp	∞	5652				**>1000%**		5652					15322			132
pooling_sppc0tp	∞	2826				∞		2826					9074			255
pooling_sppc1pq	∞	4770				∞		4770					15083			961
pooling_sppc1stp	∞	9540				**985%**		9540					25396			109
pooling_sppc1tp	∞	4770				∞		4770					15083			
pooling_sppc3pq	∞	9116				∞		9116					28416			104
pooling_sppc3stp	∞	18232				**355%**		18232					47716			0
pooling_sppc3tp	∞	9116				∞		9116					28416			
popdynm100	∞	1601				∞	1401	4400		1			9560			
popdynm200	∞	3201				∞	2801	8800		1			18760			
popdynm25	∞	401				∞	351	1100		1			2780			
popdynm50	∞	801				∞	701	2200		1			5040			
portfol_buyin	**0.4s**	1			2	**0.5s**	1			1			11			
portfol_card	0.5s	1			2	**0.3s**	1			1			11			
portfol_classical050_1	147.2s	1				**6.8s**							101			
portfol_classical200_2	16.0%	1				**8.2%**							401			
portfol_robust050_34	**5.7s**	1	50			7.2s			1				204			
portfol_robust100_09	**21.8s**	1	100			**22.7s**			1				404			
portfol_robust200_03	3.5%	1	200			**0.8%**			1				804			
portfol_roundlot	fail	1			2	**0.05%**	1			1			12			
portfol_shortfall050_68	17.2s		100			**7.6s**			2				206			
portfol_shortfall100_04	1900.9s		200			**976.1s**			2				406			
portfol_shortfall200_05	**1.3%**		400			**1.3%**			2				806			
powerflow0009r	88.4%	73				**10.2s**	33	34					215		1782	
powerflow0014r	∞	149				**0.02%**	70	76					333		21498	
powerflow0030r	∞	305				**0.54%**	160	160					902		87468	
powerflow0039r	>1000%	355				**0.23%**	186	182					1058		176622	
powerflow0057r	∞	582				**96.4s**	280	304					1287		31898	
powerflow0118r	∞	1310				**3.1%**	701	704					3088		160886	
powerflow0300r	∞	3012				∞	1514	1630					7572		150525	
powerflow2383wpr	∞	21621				∞	12092	11528					63380		22000	
powerflow2736spr	∞	26135				**39.2%**	14908	13966					74272		22000	
primary	**>1000%**	56		7	12	>1000%	14	962		13	7	9	1188	586	23249	39
prob02	**0.0s**	5				0.1s		5					11			1
prob03	**0.0s**	1				**0.0s**		1					3			
prob06	0.0s	2				**0.0s**	2						6			
prob07	357.4s	27			1	**12.7s**	5	17				17	64			
prob09	**0.0s**	2				2.2s	1	1					8			
process	**0.4s**	7		1		0.8s	3	5		1	1	2	25			
procsel	**0.0s**				2	**0.0s**				1	1		9	2		
procsyn	∞			9	10	**0.1s**							69			
procurement1large	**371%**	69			68	**373%**	1	68		68			342			
procurement1mot	**534%**	13			12	**444%**	1	12		12			62			
procurement2mot	**1.7s**				10	**1.8s**							40			
product	30.2s	82				**16.6s**	30		1				470	26		
product2	**2.7s**	128				4.5s	4						575			
prolog	∞	3				**0.1s**	1	4					16			
qap	>1000%					**465%**										
qapw	**50.8%**	1				379%										
qp2	**0.0s**	1				1.0s	1			1			51			
qp3	**58.3s**	1				∞		50					152			
qp4	**0.0s**	1				0.0s							60			
qspp_0_10_0_1_10_1	312.4s					**140.9s**										
qspp_0_11_0_1_10_1	1083.8s					**484.6s**										
qspp_0_12_0_1_10_1	23.4%					**1702.0s**										
qspp_0_13_0_1_10_1	67.3%					**50.1%**										
qspp_0_14_0_1_10_1	**99.3%**					**99.5%**										
qspp_0_15_0_1_10_1	**135%**					**155%**										
radar-2000-10-a-6_lat_7	383.8s		2000			**101.1s**			2000				12000			
radar-3000-10-a-8_lat_7	908.3s		3000			**356.1s**			3000				18000			
ramsey	0.1s				10	**0.0s**				3	1		59			
ravempb	**2.9s**				2	**2.4s**				16			63			
rbrock	**0.0s**	2				3.8s	1	1					8			
ringpack_10_1	**16.0%**	330				42.9%	185	90		40			325		211424	30
ringpack_10_2	**8.2%**	420				16.0%	230	90		40			379		84248	16
ringpack_20_1	**44.4%**	2337				218%	1246	380		155			1726		394294	532
ringpack_20_2	**34.2%**	2717				300%	1436	380		155			1935		353417	246
ringpack_20_3	**94.6%**	3055				**88.3%**	1604	380		153			2121		315832	
ringpack_30_1	**275%**	7433				436%	3888	870		343			4908		152290	30
ringpack_30_2	**219%**	8303				900%	4323	870		343			5372		135887	440
risk2bpb	0.1s				1	**0.1s**							8			
rocket100	∞	697		100	200	**1.9%**	299	403		100	100		2202			
rocket200	∞	1397		200	400	**12.6%**	599	803		200	200		4402			
rocket400	∞	2797		400	800	**>1000%**	1199	1603		400	400		8802			
rocket50	∞	347		50	100	**0.64%**	149	203		50	50		1102			
routingdelay_bigm	**25.8s**	1760				fail				127			1394	28		
routingdelay_proj	∞	3604				∞	382			6		1797	6297			
rsyn0805h	**0.2s**	9			3	1.9s	3	6		6	4	6	35			
rsyn0805hfsg	**0.3s**	3			2	3.9s	3	6		6	4	6	35			
rsyn0805m	**2.3s**				3	9.5s				3			17	2		
rsyn0805m02h	**1.0s**	6			4	14.0s	6	12		10	6	12	65			
rsyn0805m02hfsg	**1.1s**				4	4.9s				4			20			
rsyn0805m02m	**16.6s**				6	31.4s				6			33	4		

	classic					new									
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
rsyn0805m03h	**2.2s**				6	14.7s	9	18		14	8	18	95			
rsyn0805m03hfsg	**1.2s**	6			6	6.8s	9	18		14	8	18	95			
rsyn0805m03m	**7.8s**				9	40.9s				9			49	6		
rsyn0805m04h	**1.6s**	36			12	7.7s	12	24		18	10	24	125			
rsyn0805m04hfsg	**2.3s**	36			12	5.0s	12	24		18	10	24	125			
rsyn0805m04m	**16.1s**				12	39.9s				12			65	8		
rsyn0810h	**0.2s**	18			6	6.0s	6	11		11	6	11	66			
rsyn0810hfsg	**0.2s**	18			6	1.3s	4	7		8	3	7	48			
rsyn0810m	3.0s				6	**0.9s**				6			33	5		
rsyn0810m02h	**2.7s**	3			10	164.2s	12	22		20	10	22	127			
rsyn0810m02hfsg	**3.0s**	3			10	12.7s	12	22		20	10	22	127			
rsyn0810m02m	**12.0s**				12	15.6s				12			65	10		
rsyn0810m03h	**12.8s**	9			15	48.3%	18	33		29	14	33	188			
rsyn0810m03hfsg	**8.5s**	9			15	83.3s	18	33		29	14	33	188			
rsyn0810m03m	**13.9s**				18	42.1s				18			97	15		
rsyn0810m04h	**3.1s**	72			24	126.9s	24	44		38	18	44	249			
rsyn0810m04hfsg	**3.8s**	12			20	6.2s	24	44		38	18	44	249			
rsyn0810m04m	**16.8s**				24	37.9s				24			129	20		
rsyn0815h	**1.2s**	33			11	19.5s	12	21		19	9	21	117			
rsyn0815hfsg	**1.1s**	33			11	4.3s	12	21		19	9	21	117			
rsyn0815m	**1.1s**				11	1.7s				11			61	10		
rsyn0815m02h	**2.0s**	66			22	205.2s	24	42		36	16	42	229			
rsyn0815m02hfsg	**2.0s**	51			19	10.8s	24	42		36	16	42	229			
rsyn0815m02m	**12.2s**				22	**10.8s**				22			121	20		
rsyn0815m03h	**5.3s**	99			33	1481.0s	36	63		53	23	63	341			
rsyn0815m03hfsg	**5.6s**	96			33	72.1s	36	63		53	23	63	341			
rsyn0815m03m	**34.3s**				33	55.7s				33			181	30		
rsyn0815m04h	**4.1s**	60			28	17.5%	48	84		70	30	84	453			
rsyn0815m04hfsg	**7.0s**	120			44	226.4s	48	84		70	30	84	453			
rsyn0815m04m	77.5s				44	**41.5s**				44			241	40		
rsyn0820h	**1.6s**				8	21.9s	15	29		25	12	29	156			
rsyn0820hfsg	**1.2s**	42			14	3.7s	15	29		25	12	29	156			
rsyn0820m	**2.3s**				14	**2.6s**				14			76	13		
rsyn0820m02h	**2.4s**	57			25	0.67%	30	58		48	22	58	307			
rsyn0820m02hfsg	**2.4s**	54			26	38.9s	30	58		48	22	58	307			
rsyn0820m02m	**16.9s**				28	60.9s				28			151	26		
rsyn0820m03h	**7.7s**	126			42	111%	45	87		71	32	87	458			
rsyn0820m03hfsg	**9.7s**	78			38	11.1%	45	87		71	32	87	458			
rsyn0820m03m	**14.4s**				42	26.5s				42			226	39		
rsyn0820m04h	**5.4s**	129			52	128%	60	116		94	42	116	609			
rsyn0820m04hfsg	**10.5s**	162			56	14.8%	60	116		94	42	116	609			
rsyn0820m04m	**65.0s**				56	**74.4s**				56			301	52		
rsyn0830h	**3.9s**	15			10	679.9s	21	41		36	17	41	224			
rsyn0830hfsg	**4.0s**	18			11	5.6s	19	34		34	15	34	204			
rsyn0830m	**1.6s**				20	2.1s				20			109	19		
rsyn0830m02h	**6.0s**	51			27	156%	42	82		70	32	82	443			
rsyn0830m02hfsg	**4.9s**	75			34	384.1s	40	75		68	32	75	425			
rsyn0830m02m	**11.8s**				40	15.6s				40			217	38		
rsyn0830m03h	**8.8s**	180			60	95.8%	63	123		104	47	123	662			
rsyn0830m03hfsg	**9.6s**	144			57	2459.9s	59	109		100	45	109	624			
rsyn0830m03m	36.2s				60	**27.4s**				60			325	57		
rsyn0830m04h	**10.6s**	240			80	120%	84	163	1	137	62	163	876			
rsyn0830m04hfsg	**12.0s**	240			80	10.0%	80	149	1	133	58	149	836			
rsyn0830m04m	185.4s				80	**129.8s**				80			433	76		
rsyn0840h	**1.1s**	6			7	241.8s	29	58		50	23	58	313			
rsyn0840hfsg	**1.9s**	69			25	12.3s	29	58		50	23	58	313			
rsyn0840m	**1.8s**				28	2.9s				28			151	27		
rsyn0840m02h	**6.7s**	90			44	246%	58	116		98	44	116	621			
rsyn0840m02hfsg	**6.7s**	132			50	6.6%	58	116		98	44	116	621			
rsyn0840m02m	**12.8s**				56	**13.3s**				56			301	54		
rsyn0840m03h	**12.0s**	105			55	52.6%	87	174		146	65	174	929			
rsyn0840m03hfsg	**10.8s**	252			84	8.2%	87	174		146	65	174	929			
rsyn0840m03m	20.9s				84	**12.0s**				84			451	81		
rsyn0840m04h	**18.2s**	279			112	154%	116	231	1	193	86	231	1232			
rsyn0840m04hfsg	**10.2s**	336			112	24.9%	116	231	1	193	86	231	1232			
rsyn0840m04m	180.5s				112	**121.5s**				112			601	108		
saa_2	∞	23770		14		∞	2021	3610		221			13681		0	3
sambal	0.1s	1				**0.0s**							28			
sample	**0.0s**				2	**0.0s**							10			
sep1	**0.2s**	6				**0.1s**		6					23			
sepasequ_complex	**11.6%**	321		10	8	22.7%	50	413		100	100	280	1014			
sepasequ_convent	fail	205		18		fail	81	144		52	52	52	618	45		
sfacloc1_2_80	**64.0%**	60				**58.8%**							195			
sfacloc1_2_90	**22.6%**	60				**27.6%**							202			
sfacloc1_2_95	**9.3%**	39				23.3%	7	28					188			
sfacloc1_3_80	**462%**	60				>1000%							270			
sfacloc1_3_90	**350%**	60				446%							277			
sfacloc1_3_95	**158%**	39				**142%**	7	42					270			
sfacloc1_4_80	∞	75				**>1000%**							345			
sfacloc1_4_90	>1000%	75				**>1000%**							352			
sfacloc1_4_95	>1000%	47				**802%**	7	56					345			
sfacloc2_2_80	7.9s	60				**6.3s**	12						198			
sfacloc2_2_90	**2.2s**	60				**1.9s**	12						202			
sfacloc2_2_95	**1.3s**	46				**1.4s**	20						195			
sfacloc2_3_80	**29.2s**	90				48.9s	18						294			
sfacloc2_3_90	**10.5s**	90				**11.3s**	18						298			
sfacloc2_3_95	15.9s	69				**12.3s**	30						295			

	classic					new									
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
sfacloc2_4_80	**80.7s**	120				127.7s	24						384			
sfacloc2_4_90	**10.1s**	120				24.5s	24						388			
sfacloc2_4_95	26.0s	92				**10.8s**	40						385			
shiporig	∞	12		1	1	∞		3		1	1		38			
sjup2	3.8%	15120				**1098.1s**		43200					60128			
slay04h	1.4s	1				**0.9s**							18			
slay04m	1.3s	1				**0.3s**							18			
slay05h	3.9s	1				**2.6s**							22			
slay05m	0.9s	1				**0.7s**							22			
slay06h	**3.8s**	1				**3.1s**							26			
slay06m	1.7s	1				**1.4s**							26			
slay07h	9.3s	1				**2.7s**							30			
slay07m	**2.2s**	1				**2.2s**							30			
slay08h	19.0s	1				**10.9s**							34			
slay08m	**2.5s**	1				3.6s							34			
slay09h	38.0s	1				**12.0s**							38			
slay09m	**4.3s**	1				**4.0s**							38			
slay10h	493.4s	1				**176.9s**							42			
slay10m	**15.5s**	1				**18.4s**							42			
smallinvDAXr1b010-011	**0.3s**	1				1.8s	1	28					55			9
smallinvDAXr1b020-022	**0.3s**	1				4.2s	1	21					45			10
smallinvDAXr1b050-055	**0.5s**	1				**0.6s**	1			1			8			
smallinvDAXr1b100-110	**0.6s**	1				**0.7s**	1			1			8			
smallinvDAXr1b150-165	**1.4s**	1				797.3s	1	28					55			14
smallinvDAXr1b200-220	**0.9s**	1				87.9s	1	21					45			11
smallinvDAXr2b010-011	**0.3s**	1				2.0s	1	28					55			10
smallinvDAXr2b020-022	**0.3s**	1				2.6s	1	21					45			11
smallinvDAXr2b050-055	**0.5s**	1				**0.6s**	1			1			8			
smallinvDAXr2b100-110	**0.5s**	1				0.8s	1			1			8			
smallinvDAXr2b150-165	**1.4s**	1				413.3s	1	28					55			13
smallinvDAXr2b200-220	**0.9s**	1				260.2s	1	21					45			11
smallinvDAXr3b010-011	**0.3s**	1				1.4s	1	28					55			10
smallinvDAXr3b020-022	**0.3s**	1				3.6s	1	21					45			10
smallinvDAXr3b050-055	**0.5s**	1				0.6s	1			1			8			
smallinvDAXr3b100-110	**0.5s**	1				0.8s	1			1			8			
smallinvDAXr3b150-165	**1.3s**	1				291.1s	1	28					55			12
smallinvDAXr3b200-220	**0.8s**	1				201.2s	1	21					45			10
smallinvDAXr4b010-011	**0.3s**	1				1.5s	1	28					55			11
smallinvDAXr4b020-022	**0.3s**	1				7.0s	1	21					45			10
smallinvDAXr4b050-055	0.8s	1				**0.6s**	1			1			8			
smallinvDAXr4b100-110	**0.5s**	1				2.0s	1			1			8			
smallinvDAXr4b150-165	**1.2s**	1				189.2s	1	21					45			10
smallinvDAXr4b200-220	0.9s	1				**0.5s**	1			1			9			
smallinvDAXr5b010-011	**0.3s**	1				5.2s	1	28					55			12
smallinvDAXr5b020-022	**0.3s**	1				3.3s	1	21					45			11
smallinvDAXr5b050-055	**0.8s**	1				1.6s	1			1			8			
smallinvDAXr5b100-110	**0.5s**	1				0.8s	1			1			8			
smallinvDAXr5b150-165	**1.1s**	1				**1.3s**	1			1			8			
smallinvDAXr5b200-220	**0.7s**	1				2.6s	1			1			9			
smallinvSNPr1b010-011	**6.1s**	1				12.9s	1	4950					5152	275		14
smallinvSNPr1b020-022	95.8s	1				**44.8s**	1	4950					5152	1287		12
smallinvSNPr1b050-055	∞	1				**166.4s**	1	4950					5152	3521		16
smallinvSNPr1b100-110	∞	1				**52.8s**	1	4950					5152	3756		19
smallinvSNPr1b150-165	∞	1				**89.8s**	1	4950					5152	10404		21
smallinvSNPr1b200-220	234%	1				**6.4%**	1	4950					5152	54763		15
smallinvSNPr2b010-011	**6.6s**	1				12.0s	1	4950					5152	261		16
smallinvSNPr2b020-022	**30.8s**	1				**29.9s**	1	4950					5152	1130		14
smallinvSNPr2b050-055	3290.8s	1				**40.0s**	1	4950					5152	2493		15
smallinvSNPr2b100-110	∞	1				**78.0s**	1	4950					5152	7034		27
smallinvSNPr2b150-165	∞	1				**1645.2s**	1	4950					5152	45322		19
smallinvSNPr2b200-220	∞	1				**9.5%**	1	4950					5152	76375		14
smallinvSNPr3b010-011	**4.3s**	1				11.0s	1	4560					4949	189		30
smallinvSNPr3b020-022	**20.9s**	1				80.5s	1	4950					5152	1172		12
smallinvSNPr3b050-055	**104.1s**	1				**96.8s**	1	4950					5152	8025		12
smallinvSNPr3b100-110	1268.4s	1				**185.2s**	1	4950					5152	11081		16
smallinvSNPr3b150-165	2900.5s	1				**781.5s**	1	4950					5152	31943		19
smallinvSNPr3b200-220	211%	1				**634.5s**	1	4950					5152	26307		17
smallinvSNPr4b010-011	**4.0s**	1				16.7s	1	4950					5152	262		13
smallinvSNPr4b020-022	**12.4s**	1				20.8s	1	4950					5152	775		9
smallinvSNPr4b050-055	**35.6s**	1				**29.4s**	1	4950					5152	1181		18
smallinvSNPr4b100-110	201.2s	1				**127.0s**	1	4950					5152	6516		17
smallinvSNPr4b150-165	59.2%	1				**874.9s**	1	4950					5152	27915		17
smallinvSNPr4b200-220	2135.6s	1				**242.3s**	1	4950					5152	16514		16
smallinvSNPr5b010-011	**3.2s**	1				12.0s	1	4465					4948	113		34
smallinvSNPr5b020-022	**4.6s**	1				15.9s	1	4950					5152	471		11
smallinvSNPr5b050-055	42.1s	1				**23.6s**	1	4950					5152	1118		14
smallinvSNPr5b100-110	**69.7s**	1				**81.5s**	1	4950					5152	5503		16
smallinvSNPr5b150-165	138.5s	1				**51.7s**	1	4950					5152	4752		25
smallinvSNPr5b200-220	**427.0s**	1				1046.2s	1	4950					5152	26806		61
sonet17v4	**924.7s**					**1090.8s**										
sonet18v6	**950.0s**					1367.4s										
sonet19v5	**12.1%**					15.7%										
sonet20v6	2000.0s					**1001.0s**										
sonet21v6	**6%**					**7.3%**										
sonet22v4	11.6%					**8.6%**										
sonet22v5	**108%**					**125%**										
sonet23v4	**48.1%**					**44.0%**										

	classic						new										
instance	time/gap	quad	soc	abspow	nonlin		time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
sonet23v6	**24.6%**						**20.9%**										
sonet24v2	**191.8s**						320.1s										
sonet24v5	**91.9%**						137%										
sonet25v5	**12.4%**						**12.9%**										
sonet25v6	**188%**						249%										
sonetgr17	**49.6s**						982.9s										
space25	∞	25					∞	25	76					152			
space25a	∞	25					∞	25	86					179			
space960	∞	960					∞	952	3740					7357			
spectra2	**2.1s**	8					10.2s	8	26		8			68	13	186	
sporttournament06	0.0s						**0.0s**										
sporttournament08	**0.1s**						0.3s										
sporttournament10	**0.3s**						**0.3s**										
sporttournament12	**0.2s**						**0.2s**										
sporttournament14	9%	1					**0.6s**										
sporttournament16	**0.5s**						1.2s										
sporttournament18	**5.3s**						**5.7s**										
sporttournament20	**19.7s**						34.1s										
sporttournament22	66.1s						**39.9s**										
sporttournament24	**77.5s**						164.8s										
sporttournament26	995.2s						**677.1s**										
sporttournament28	**207.6s**						356.5s										
sporttournament30	1.6%						**1.2%**										
sporttournament32	**1.6%**						**1.6%**										
sporttournament34	**1.8%**						**2%**										
sporttournament36	**1.9%**						**2.3%**										
sporttournament38	**3.7%**						**3.9%**										
sporttournament40	10.0%						**6.9%**										
sporttournament42	7.4%						**5.4%**										
sporttournament44	8.8%						**6.3%**										
sporttournament46	**7.1%**						**7.7%**										
sporttournament48	13.1%						**7.6%**										
sporttournament50	**9.3%**						**8.4%**										
spring	**0.2s**	9		2	2		0.3s	2	3		1	1	2	26			
squfl010-025	1278.0s	1					**3.1s**							502	250		
squfl010-025persp	1.4s		250				**0.5s**			250				1260			
squfl010-040	43.7%	1					**2.5s**							802	400		
squfl010-040persp	3.5s		400				**0.6s**			400				2010			
squfl010-080	132%	1					**35.2s**							1602	800		
squfl010-080persp	98.0s		800				**7.8s**			560				3767			
squfl015-060	112%	1					**8.1s**							1802	900		
squfl015-060persp	78.7s		900				**5.2s**			720				4332			
squfl015-080	117%	1					**114.9s**							2402	1200		
squfl015-080persp	575.1s		1200				**10.2s**			800				5610			
squfl020-040	80.2%	1					**6.7s**							1602	800		
squfl020-040persp	21.2s		800				**2.6s**			800				4020			
squfl020-050	118%	1					**56.6s**							2002	1000		
squfl020-050persp	300.9s		1000				**6.5s**			700				4714			
squfl020-150	470%	1					**1605.7s**							6002	3000		
squfl020-150persp	2247.7s		3000				**62.7s**			1950				13963			
squfl025-025	57.8%	1					**4.7s**							1252	625		
squfl025-025persp	28.6s		600				**5.7s**			500				2820			
squfl025-030	53.0%	1					**33.2s**							1502	750		
squfl025-030persp	65.4s		720				**14.8s**			570				3229			
squfl025-040	166%	1					**48.4s**							2002	1000		
squfl025-040persp	0%		1000				**63.2s**			800				4820			
squfl030-100	266%	1					**878.8s**							6002	3000		
squfl030-100persp	**0.01%**		3000				**0.01%**			1800				12218			
squfl030-150	577%	1					**112.5s**							9002	4500		
squfl030-150persp	0.23%		4500				**0.01%**			3150				21171			
squfl040-080	291%	1					**31.0s**							6402	3200		
squfl040-080persp	**0.01%**		3200				**0.01%**			2480				15311			
srcpm	**0.0s**				1		0.1s							12			
sssd08-04	15.1s	4					**1.3s**						4	12			
sssd08-04persp	72.6s	12					**17.8s**	12	12					76			3
sssd12-05	6.1%	5					**1.4s**						5	19			
sssd12-05persp	**7%**	15					**7%**	15	15					95			3
sssd15-04	14.0%	4					**1.1s**						4	12			
sssd15-04persp	35.5%	12					**11.8%**	12	12					76			1
sssd15-06	20.3%	6					**14.1s**						6	34			
sssd15-06persp	71.5%	18					**25.7%**	18	18					114			5
sssd15-08	17.8%	8					**21.1s**						8	24			
sssd15-08persp	75.6%	24					**19.5%**	24	24					152			4
sssd16-07	17.6%	7					**57.9s**						8	29			
sssd16-07persp	68.8%	21					**18.6%**	21	21					133			5
sssd18-06	16.5%	6					**7.1s**						6	36			
sssd18-06persp	67.9%	18					**18.7%**	18	18					114			4
sssd18-08	28.9%	8					**84.1s**						8	56			
sssd18-08persp	101%	24					**34.2%**	24	24					152			7
sssd20-04	18.4%	4					**1.5s**						4	12			
sssd20-04persp	59.0%	12					**20.1%**	12	12					76			4
sssd20-08	16.1%	8					**203.9s**						8	56			
sssd20-08persp	68.2%	24					**19.7%**	24	24					152			6
sssd22-08	17.8%	8					**110.5s**						8	56			
sssd22-08persp	74.7%	24					**20.5%**	24	24					152			6
sssd25-04	23.8%	4					**2.1s**						4	28			
sssd25-04persp	77.8%	12					**23.7%**	12	12					76			4

	classic					new									
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
sssd25-08	14.7%	8				**114.2s**						9	38			
sssd25-08persp	65.3%	24				**16.0%**	24	24					152			4
st_bpaf1a	**0.0s**	1				0.0s		5					17			
st_bpaf1b	**0.0s**	1				0.0s		5					17			
st_bpk1	0.0s	1				**0.0s**	1	4					10			0
st_bpv1	**0.0s**					**0.0s**										
st_bpv2	**0.0s**	1				**0.0s**										
st_bsj2	0.1s	1				**0.1s**							8			
st_bsj3	**0.0s**	1				**0.0s**							10			
st_bsj4	**0.1s**	1				**0.1s**							14			
st_cqpf	0.0s	1				**0.0s**							10			
st_cqpjk1	0.0s	1				**0.0s**							10			
st_cqpjk2	0.0s	1				**0.0s**							8			
st_e01	**0.0s**	1				**0.0s**		1					3			
st_e02	**0.0s**	3				**0.0s**	3	1		1			9			
st_e03	**29.8s**	7		2		0.07%	3	4					22			
st_e04	0.6s	1			3	**0.2s**		1		3	2		16			
st_e05	0.1s	2				**0.1s**	2	2	1				8			
st_e06	**0.0s**				1	**0.0s**	1						4			
st_e07	0.1s	3				**0.0s**	1	2					14			
st_e08	**0.0s**	2				**0.0s**		1					7		0	
st_e09	0.1s	2				**0.0s**	1	2					7			
st_e11	**0.1s**	1			1	0.1s	1	1					8			
st_e12	**0.0s**				1	**0.0s**							6			
st_e13	**0.0s**			1		**0.0s**							4			
st_e14	0.1s	1		2	1	**0.1s**	1			1			18			
st_e15	**0.0s**			2		**0.0s**							8	2		
st_e16	0.1s	6			1	**0.1s**	4	8		4			29			
st_e17	**0.0s**				1	0.0s				1		1	6			
st_e18	**0.0s**	2				**0.0s**							5			
st_e19	0.2s	1			1	**0.1s**	2						8			
st_e21	**0.1s**				1	**0.1s**							8			
st_e22	**0.0s**	1				**0.0s**							6			
st_e23	0.1s	1				**0.0s**		1					5			
st_e24	0.0s	1				**0.0s**							6			
st_e25	**0.1s**	1				0.2s	1	6					16		32	0
st_e26	0.0s	1				**0.0s**							6			
st_e27	**0.0s**	1				**0.0s**							6	2		
st_e28	**0.0s**	4				0.0s	3	7					22			
st_e29	0.1s	2			3	**0.0s**				3	2		23			
st_e30	**0.3s**	5				**0.3s**		6					24			
st_e31	2.4s	5				**1.5s**		6					24			
st_e32	**3.5s**	25		5	6	6.4s	6	165		9	4	8	264		130	1
st_e33	0.1s	3				**0.0s**	1	4					16			
st_e34	0.0s	4				**0.0s**	2	5					13			
st_e35	**10.4s**	19		6	8	108%	12	19		7	5		82	6	166933	0
st_e36	0.8s	9		2	1	**0.3s**	7	1		5			17		10	
st_e37	**0.6s**	1			20	1.2s				30	20		104			
st_e38	0.1s	3		2		**0.1s**	1	3					13			
st_e40	**0.1s**	16				0.2s	1	3					32			
st_e41	**0.1s**	7		3	1	1.5s	4	4					36		2	
st_e42	**0.0s**	1				**0.0s**	1	1					8			
st_fp7a	**0.2s**	1				0.3s							42			
st_fp7b	**0.3s**	1				0.5s							42			
st_fp7c	**0.2s**	1				0.3s							42			
st_fp7d	**0.2s**	1				**0.3s**							42			
st_fp7e	**0.3s**	1				1.6s							42			
st_fp8	**0.1s**	1				**0.1s**							50			
st_glmp_fp1	**0.0s**	1				**0.0s**	1						6			
st_glmp_fp2	0.0s	1				**0.0s**	1						6			
st_glmp_fp3	**0.0s**	1				**0.0s**							6			
st_glmp_kk90	**0.0s**	1				**0.0s**	1						6			
st_glmp_kk92	0.0s	1				**0.0s**							6			
st_glmp_kky	**0.0s**	1				**0.0s**	1						6			
st_glmp_ss1	0.1s	1				**0.1s**	1						6			
st_glmp_ss2	**0.0s**	1				0.0s	1	1					7			0
st_ht	**0.0s**	1				0.1s							6			
st_iqpbk1	**0.2s**	1				**0.2s**	1	28					46		56	0
st_iqpbk2	**0.3s**	1				**0.2s**	1	28					46		107	0
st_jcbpaf2	**0.1s**	1				0.1s		5					17			
st_m1	**0.2s**	1				0.3s							42			
st_m2	**0.2s**	1				0.5s							62			
st_miqp1	**0.0s**					**0.0s**										
st_miqp2	**0.0s**	1				**0.0s**							6	2		
st_miqp3	**0.0s**	1				**0.0s**							4			
st_miqp4	**0.0s**	1				0.0s							8	2		
st_miqp5	0.0s	1				**0.0s**							6			
st_pan1	**0.1s**	1				**0.1s**							8			
st_ph1	0.0s	1				**0.0s**							14			
st_ph10	**0.0s**	1				**0.0s**	1						4			
st_ph11	**0.0s**	1				0.1s							8			
st_ph12	**0.0s**	1				**0.1s**							8			
st_ph13	**0.0s**	1				0.1s							8			
st_ph14	**0.0s**	1				**0.0s**							8			
st_ph15	**0.0s**	1				0.1s							10			
st_ph2	**0.0s**	1				**0.0s**							14			
st_ph20	**0.0s**	1				0.1s							6			

	classic					new									
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
st_ph3	**0.0s**	1				0.0s							10			
st_phex	0.0s	1				**0.0s**							6			
st_qpc-m0	**0.0s**	1				**0.0s**							6			
st_qpc-m1	**0.1s**	1				0.1s	1	10					22		17	0
st_qpc-m3a	**0.2s**	1				**0.3s**	1	45					67		29	0
st_qpc-m3b	**0.1s**	1				0.2s	1	45					67		60	0
st_qpc-m3c	**0.0s**	1				**0.0s**	1	45					67			
st_qpc-m4	**0.0s**	1				0.0s	1	44					66			
st_qpk1	**0.0s**	1				0.1s	1	1					7		0	0
st_qpk2	**0.1s**	1				**0.1s**	1	5					19			
st_qpk3	**0.2s**	1				**0.2s**	1	10					34		0	
st_robot	**0.0s**	7				0.0s	3	2					23		15	
st_rv1	**0.1s**	1				**0.2s**							22			
st_rv2	**0.2s**	1				**0.2s**							42			
st_rv3	**0.3s**	1				0.6s							42			
st_rv7	**0.3s**	1				1.4s							62			
st_rv8	**0.5s**	1				1.5s							82			
st_rv9	**1.3s**	1				3.4s							102			
st_test1	**0.0s**					**0.0s**										
st_test2	**0.0s**					**0.0s**										
st_test3	**0.0s**					**0.0s**										
st_test4	**0.0s**	1				**0.0s**							6			
st_test5	**0.0s**					**0.0s**										
st_test6	**0.0s**					**0.0s**										
st_test8	**0.0s**	1				0.0s							50			
st_testgr1	**0.1s**	1				**0.1s**							12			
st_testgr3	**0.1s**	1				0.2s							14			
st_testph4	**0.0s**	1				**0.0s**							6			
st_z	**0.1s**	1				**0.1s**							8			
steenbrf	∞	108			37	**30.4s**	1	36		36			434			
stockcycle	87.1s				1	**0.7s**				13			49			
super1	fail	378		44	236	**334%**	274	841	4	46	46	125	2174	18	53857	2
super2	fail	381		44	236	**348%**	277	840	6	46	46	125	2176	18	17482	2
super3	fail	385		44	236	**391%**	285	847	6	46	46	125	2202	19	33706	3
super3t	**49.1%**	237		32	163	**53.4%**	189	724	2	19	19	125	1592	14	8635	7
supplychain	**0.3s**			6		**0.4s**							36			
supplychainp1_020306	0.2s				1	**0.1s**					7		23			
supplychainp1_022020	**1099.3s**				1	**1184.5s**					20		502			
supplychainp1_030510	**1.7s**				1	4.8s					10		92	5		
supplychainp1_053050	**43.0%**				1	**50.1%**					50		1712			
supplychainr1_020306	0.3s				1	**0.1s**					7		23			
supplychainr1_022020	0.24%				1	**32.5s**					30		92	4		
supplychainr1_030510	**0.1s**				1	0.2s					13		41	2		
supplychainr1_053050	16.2%				1	**1.5%**					80		242			
syn05h	**0.1s**	9			3	0.1s	2	4		5	3	4	28			
syn05hfsg	**0.1s**	9			3	0.2s	1	2		2	1	2	12			
syn05m	0.2s				2	**0.0s**				1			6	1		
syn05m02h	**0.1s**	18			6	0.3s	6	12		10	6	12	65			
syn05m02hfsg	**0.1s**	18			6	0.2s	5	10		9	6	10	59			
syn05m02m	**0.1s**				4	**0.1s**				6			33	4		
syn05m03h	**0.1s**	27			9	0.4s	8	16		13	7	16	88			
syn05m03hfsg	**0.1s**	27			9	0.5s	7	14		12	6	14	81			
syn05m03m	**0.2s**				9	**0.2s**				9			49	6		
syn05m04h	**0.1s**	36			12	0.3s	11	22		17	9	22	118			
syn05m04hfsg	**0.1s**	36			12	0.2s				8			40			
syn05m04m	0.3s				12	**0.2s**				12			65	8		
syn10h	**0.1s**	12			5	0.4s	6	11		11	6	11	66			
syn10hfsg	**0.1s**	18			6	0.3s	6	11		11	6	11	66			
syn10m	0.1s				6	**0.0s**				6			33	5		
syn10m02h	**0.3s**	36			12	1.2s	12	22		20	10	22	127			
syn10m02hfsg	**0.2s**	36			12	1.1s	12	22		20	10	22	127			
syn10m02m	0.7s				12	**0.4s**				10			49	10		
syn10m03h	**0.5s**	54			18	4.6s	18	33		29	14	33	188			
syn10m03hfsg	**0.3s**	54			18	1.9s	18	33		29	14	33	188			
syn10m03m	1.0s				12	**0.6s**				15			81	15		
syn10m04h	**0.6s**	72			24	17.2s	24	44		38	18	44	249			
syn10m04hfsg	**0.4s**	72			24	2.1s	22	41		37	17	41	239			
syn10m04m	1.6s				24	**0.7s**				24			129	20		
syn15h	**0.1s**	33			11	0.6s	11	20		19	9	20	116			
syn15hfsg	**0.1s**	33			11	0.8s	11	20		19	9	20	116			
syn15m	**0.1s**				11	0.2s				11			60	10		
syn15m02h	**0.3s**	66			22	4.2s	22	40		36	16	40	227			
syn15m02hfsg	**0.2s**	66			22	2.1s	22	40		36	16	40	227			
syn15m02m	**0.3s**				22	**0.3s**				22			117	20		
syn15m03h	**0.4s**	99			33	9.2s	33	60		53	23	60	338			
syn15m03hfsg	**0.4s**	99			33	4.9s	32	58		52	22	58	331			
syn15m03m	**0.5s**				33	**0.6s**				33			175	30		
syn15m04h	**0.5s**	132			44	24.8s	44	80		70	30	80	449			
syn15m04hfsg	**0.7s**	132			44	6.3s	44	80		70	30	80	449			
syn15m04m	**1.2s**				44	**1.3s**				44			232	40		
syn20h	**0.3s**	33			13	3.1s	14	28		25	12	28	155			
syn20hfsg	**0.3s**	33			13	1.5s	14	28		25	12	28	155			
syn20m	**0.5s**				14	**0.5s**				11			56	11		
syn20m02h	**0.8s**	84			28	43.7s	28	56		48	22	56	305			
syn20m02hfsg	**0.6s**	84			28	7.0s	28	56		48	22	56	305			
syn20m02m	**0.9s**				26	**1.1s**				28			149	26		
syn20m03h	**0.8s**	126			42	490.3s	42	84		71	32	84	455			

	classic					new									
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
syn20m03hfsg	**0.6s**	126			42	17.5s	42	84		71	32	84	455			
syn20m03m	**1.6s**				27	**1.5s**				42			223	39		
syn20m04h	**1.1s**	168			56	2476.9s	56	112		94	42	112	605			
syn20m04hfsg	**1.1s**	168			56	89.1s	56	112		94	42	112	605			
syn20m04m	**2.4s**				52	**2.7s**				56			297	52		
syn30h	**0.8s**	54			20	51.1s	21	41		36	17	41	224			
syn30hfsg	fail	60			20	**1.9s**	19	34		34	15	34	204			
syn30m	**0.8s**				20	**0.7s**				20			108	19		
syn30m02h	**1.8s**	120			40	78.3%	40	80		70	32	80	441			
syn30m02hfsg	fail	120			40	**7.0s**	38	73		68	32	73	423			
syn30m02m	2.5s				22	**1.6s**				40			210	38		
syn30m03h	**3.6s**	180			60	136%	60	120		104	47	120	659			
syn30m03hfsg	**1.8s**	180			60	462.4s	56	106		100	45	106	621			
syn30m03m	**2.5s**				60	**2.7s**				60			317	57		
syn30m04h	**3.1s**	240			80	196%	80	159	1	137	62	159	872			
syn30m04hfsg	**3.8s**	240			80	10.5%	75	141	1	132	58	141	822			
syn30m04m	6.0s				80	**3.6s**				80			421	76		
syn40h	**0.8s**	84			28	237.5s	28	57		50	23	57	312			
syn40hfsg	fail	84			28	**2.2s**	28	57		50	23	57	312			
syn40m	**0.9s**				28	1.2s				28			149	27		
syn40m02h	**3.3s**	168			56	228%	56	114		98	44	114	619			
syn40m02hfsg	fail	168			56	**277.9s**	53	103		95	42	103	589			
syn40m02m	4.6s				56	**2.7s**				56			292	54		
syn40m03h	**3.4s**	252			84	382%	84	171		146	65	171	926			
syn40m03hfsg	**3.1s**	252			84	34.1%	84	171		146	65	171	926			
syn40m03m	**3.4s**				84	4.6s				84			443	81		
syn40m04h	**7.2s**	336			112	369%	112	227	1	193	86	227	1228			
syn40m04hfsg	**6.8s**	336			112	50.8%	109	216	1	190	83	216	1197			
syn40m04m	**5.8s**				112	**5.7s**				112			581	108		
synheat	21.1%	29			12	**0.4%**	5	11		18	2	7	94	6		
synthes1	0.1s				3	**0.0s**				2			11	1		
synthes2	**0.0s**				4	0.0s				2			14	1		
synthes3	0.4s				5	**0.1s**				4			22	2		
tanksize	**3.2s**	12			1	**3.4s**	6	11					59			5
telecomsp_metro	**24.9%**					48.3%										
telecomsp_njlata	**7.6%**					12.0%										
telecomsp_nor_sun	∞					∞										
telecomsp_pacbell	**3285.4s**					1.8%										
tln12	974%	12				**8.4%**		132					312			14
tln2	**0.0s**	2				0.0s		4					12			0
tln4	1.6s	4				**1.1s**		16					40			4
tln5	89.5s	5				**0.7s**		25					60			5
tln6	79.1%	6				**3.6s**		36					84			8
tln7	403%	7				**465.3s**		49					112			8
tloss	**0.0s**	6				0.7s		36					84	24		5
tls12	∞	144			12	∞	144			144			897			15
tls2	**0.2s**	4			2	0.7s	4			4			25			
tls4	**21.3s**	15			4	55.3s	16			16			165			14
tls5	64.4%	21			5	**49.3%**	25			25			261			12
tls6	196%	32			6	**67.2%**	36			36			327			10
tls7	>1000%	49			7	**>1000%**	49			49			509			8
tltr	**0.1s**	3				0.3s		9					44	9		
topopt-cantilever_60x40_50	∞		31200			**>1000%**			2399				67198			
topopt-mbb_60x40_50	∞		31200			∞			2399				67198			
topopt-zhou-rozvany_75	3.5%		1300			**2.6%**			98				2796			
toroidal2g20_5555	**5.4s**					**5.2s**										
toroidal3g7_6666	**93.8s**					**114.5s**										
torsion100	**27.6s**	2				∞	2	9850		2			19854		35545	
torsion25	**0.9s**	2				6.2%	2	2425		2			4929		16058	
torsion50	**4.1s**	2				∞	2	4900		2			9904		41426	
torsion75	**12.4s**	2				56.8%	2	7375		2			14879		28097	
trainf	∞	10000			2	∞	10001	10000					60003			
transswitch0009r	346%	97				**4.2%**	34	10					232		11886	
transswitch0014r	∞	229				∞	66						405	1		
transswitch0030r	∞	457				∞	162	12					1031	1	39801	
transswitch0039r	∞	531				**55.0%**	192	8					1198	6	1159	
transswitch0057r	∞	890				∞	272	4					1608		190	
transswitch0118r	∞	2018				∞	691	8					3879	3	1127	
transswitch0300r	∞	4488				∞	1524	156					9127	27	2829	
transswitch2383wpr	∞	31565				∞	12478	1612					71643		5130	
transswitch2736spr	**39.2%**	39151				∞	15325	972					85370		2557	
tricp	∞	190				**3250.6s**	190	462		3			869		23560	
tspn05	**332.1s**	25			10	10.3%	15			10			57			
tspn08	**8.2%**	64			28	23.6%	36			28			126			
tspn10	**5.8%**	100			45	137%	55			45			187			
tspn12	**48.7%**	144			66	168%	78			66			260			
tspn15	**32.8%**	225			105	104%	120			105			392			
turkey	**0.3s**	1				0.6%	1			1			111			
unitcommit1	4.8s	1				**2.7s**							372	111		
unitcommit2	4.3%	1			166	**14.8s**	1						873	158		271
unitcommit_200_0_5_mod_7	7.6%		4646			**0.39%**	103		4374				26920	45		
unitcommit_200_100_1_mod_8	0.13%	1				**0.05%**	1						9239	4341		
unitcommit_200_100_2_mod_7	0.25%		4639			**0.05%**	114		4400				26900	38		
unitcommit_200_100_2_mod_8	0.05%	1				**0.02%**	1						9166	4324		
unitcommit_50_20_2_mod_8	0.02%	1				**58.3s**	1						1934	498		
uselinear	fail					∞	721	4924	68	1144	1144		18178			
util	**0.2s**	4				**0.2s**	4	5					36			1

	classic					new										
instance	time/gap	quad	soc	abspow	nonlin	time/gap	quad	bilin	soc	convex	concave	quot	def	persp	minor	rlt
wager	**0.5s**	63	2			2.5s	33		1	1	6	6	277			
wall	∞	8				fail		2					14			
wallfix	0.1s	6			2	**0.1s**		2					14			
waste	85.2%	314				**826.1s**	36	1230					2526			20
wastepaper3	**9.8s**	16			6	**8.9s**		6					128			7
wastepaper4	**196.0s**	20			8	296.3s		8					192			9
wastepaper5	fail	24			10	∞		10					268			6
wastepaper6	fail	28			12	∞		12					356			6
wastewater02m1	0.2s	3				**0.1s**	2	8					28			0
wastewater02m2	**0.2s**	12				**0.2s**		12					46			0
wastewater04m1	**0.3s**	6				**0.3s**	4	16					41			0
wastewater04m2	**0.3s**	18				**0.4s**		18					66			0
wastewater05m1	fail					**9.1s**	9	45					94			9
wastewater05m2	55.1s	48				**14.6s**		48					168			0
wastewater11m1	24.5%	8				**106.9s**	7	63					182			12
wastewater11m2	fail					**1.5%**		112					406			0
wastewater12m1	28.6%	11				**138.0s**	10	120					317			6
wastewater12m2	**9.8%**	220				14.4%		220					790			0
wastewater13m1	fail					**11.1%**	15	255					638			11
wastewater13m2	**18.5%**	480				26.7%		480					1710			0
wastewater14m1	fail					**0.46%**	10	70					146			2
wastewater14m2	**0.29%**	90				1.3%		90					315			0
wastewater15m1	549.4s	12				**23.4s**	9	45					94			18
wastewater15m2	**69.1s**	48				**79.2s**		48					168			0
water	542%	29			29	**229%**	1	29		1	1		128			
water3	**154%**	29				290%	1	2					126	14		
water4	fail	29				fail	1	2					126	14		
watercontamination0202	167.4s	1				**12.3s**	1			1			878	273		
watercontamination0202r	∞	1				∞	1			1			95			
watercontamination0303	2999.3s	1				**29.1s**	1			1			1882	627		
watercontamination0303r	∞	1				∞	1			1			186			
waterful2	**372%**	57		56		∞	1	58					350	112		
waternd1	**9.3s**	12			1	**9.8s**		28			20		111	16		31
waternd2	5.3%	30			1	**1.2%**		168			70		435	64		17
waternd_blacksburg	**6.1%**	23		48		**6%**	23	44					193			
waternd_fossiron	∞	58		116		**69.8%**	58	115					441			
waternd_fosspoly0	**11.6%**	58		116		∞	58	115					441			
waternd_fosspoly1	**12.8%**	58		116		**10.4%**	58	115					439			
waternd_hanoi	**258.9s**	34		59		**271.9s**	32	61		8	8		241			
waternd_modena	∞	317		615		∞	317	630					2445			
waternd_pescara	∞	98		191		∞	98	192					741			
waternd_shamir	25.4s	8		16		**7.0s**	8	15					59			
waterno1_01	**0.9s**	8		13		**0.9s**	6	10		1	1		141	17		
waterno1_02	**1.8s**	48		31		5.5s	12	20		1	1		288	36		
waterno1_03	**16.6s**	68		47		26.5s	17	32		1	1		447	56		
waterno1_04	**27.9s**	90		63		1411.7s	23	43		1	1		594	75		
waterno1_06	**209.5s**	144		95		576.6s	36	72		1	1		960	124		
waterno1_09	**754.8s**	216		143		2263.0s	54	108		1	1		1440	187		
waterno1_12	**2.8%**	288		191		7.7%	72	144		1	1		1920	250		
waterno1_18	**16.7%**	432		287		**18.3%**	108	216		1	1		2880	376		
waterno1_24	**23.6%**	576		382		**19.9%**	144	288		1	1		3838	501		
waterno2_01	**0.4s**	27		23		0.9s	1	21					169	26	6	0
waterno2_02	**1.9s**	64		54		fail	1	45					369	79	16	4
waterno2_03	**9.1%**	108		87		fail	1	81					652	139	20	5
waterno2_04	**1235.3s**	144		116		fail	1	108					876	185	18	5
waterno2_06	326%	216		174		**128%**	1	162					1312	277	36	13
waterno2_09	>1000%	324		261		**321%**	1	243					1944	415	168	8
waterno2_12	>1000%	432		348		**571%**	1	324					2624	553	182	
waterno2_18	>1000%	648		522		**638%**	1	486					3924	829	321	1
waterno2_24	>1000%	864		696		**750%**	1	648					5236	1105	505	
waters	341%	29				**247%**	1	2					126			
watersbp	**116%**	29				412%	1	2					126	14		
watersym1	**124%**	29		28		**134%**	1	30					182	56		
watersym2	91.3%	27		26		**48.9%**	2	29					167	48		
watertreatnd_conc	1983.6s	24			5	**3.9s**	20	140					264	5		67
watertreatnd_flow	**35.2s**	150			5	**37.9s**		150					520	5		7
waterund01	0.21%	14				**1658.0s**	6	36	6				81			2080
waterund08	fail	37				**13.8s**	16	100	15				200			0
waterund11	0.12%	28				**291.4s**	12	72	4				159			107
waterund14	1%	66				**0.11%**	24	216	24				397			2322
waterund17	0.49%	27				**0.12%**	12	84	11				179			19572
waterund18	0.29%	28				**0.1%**	12	69	12				151			10265
waterund22	2.5%	66				**0.01%**	24	208	9				408			364
waterund25	3.3%	36				**1.5%**	15	135	6				283			2199
waterund27	6.9%	96				**4.9%**	32	576	18				1080			1875
waterund28	7.8%	240				**3%**	120	2640	105				3620			329
waterund32	126%	160				**29.8%**	80	1760	35				2560			522
waterund36	9.3%	110				**3.8%**	45	657	25				1072			5124
waterx	18.0%	29			15	**9.8%**	15	30					156	16		
waterz	>1000%	29				**233%**	1	2					126			
weapons	1.1%	27			65	**0.0s**				20			107			
Usage count (#instances: 1678):		1200	43	205	546		918	920	129	529	324	256	1435	250	162	266

B Detailed Computational Results to Section 10.2 on Strong Branching in GCG

The following table gives the time in seconds needed to solve each problem instance with original variable branching. Entries that perform better than `pseudocost` are *italic*, the best entry in each row is in **bold face**.

	pseudo-cost	random	most-frac	SBw/oCG	SBw/CG	hierar-chical	hybrid	reli-able	rel. hier.	hybrid hier.
12Cap10	237.7	3433.9	841.7	2071.0	1245.0	*223.7*	635.1	3197.4	**126.9**	*174.8*
14Cap10	100.9	*60.5*	176.4	452.1	401.3	**59.3**	213.9	*60.2*	135.7	*93.2*
20Cap10	171.2	613.6	402.4	1569.5	1043.5	*117.0*	458.3	625.8	*129.5*	**87.7**
NU_1_0010_05_3	**40.9**	41.2	41.7	44.5	48.5	47.2	47.5	46.7	41.4	45.1
NU_1_0010_05_7	**23.5**	49.4	23.7	36.7	30.4	181.5	27.1	27.6	40.4	32.4
NU_3_0010_05_1	8.8	29.7	9.2	17.6	11.7	**7.8**	13.2	30.1	12.8	12.5
NU_3_0010_05_3	20.5	63.3	*19.6*	60.2	26.1	23.8	29.7	63.9	21.5	**13.6**
NU_3_0010_05_5	17.3	28.9	18.7	27.4	**4.9**	*12.0*	*8.5*	*6.5*	*9.7*	*13.1*
NU_3_0010_05_7	**19.0**	19.3	19.6	19.7	25.9	30.1	27.8	19.6	20.5	23.1
NU_3_0010_05_9	10.2	26.5	16.3	19.1	11.0	13.9	14.0	26.2	15.2	**8.0**
U_1_0050_05_0	**5.3**	5.7	5.4	5.6	5.5	5.4	5.9	5.7	5.7	5.7
U_1_0050_05_2	150.1	210.8	**67.0**	486.4	*147.4*	*128.1*	*95.6*	212.1	**19.0**	297.1
U_1_0050_25_7	>3600.0	*135.9*	>3600.0	>3600.0	*707.9*	*1019.8*	*140.5*	*137.1*	>3600.0	**48.1**
U_1_0100_05_1	372.1	*277.5*	563.9	*279.4*	*278.1*	*206.8*	*136.3*	*281.1*	*186.5*	**11.6**
U_1_0100_05_3	2653.9	*2618.3*	2658.0	2667.3	*2608.4*	**2602.4**	*2653.4*	*2611.4*	*2640.6*	*2632.0*
U_2_0050_05_4	305.7	**48.2**	*158.6*	*251.2*	*164.8*	*198.4*	1344.1	*102.6*	*164.0*	*74.9*
U_2_0100_05_2	>3600.0	*847.0*	>3600.0	*846.4*	>3600.0	>3600.0	>3600.0	**844.2**	>3600.0	>3600.0
U_2_0100_05_5	623.7	*382.0*	>3600.0	*455.2*	*378.1*	1582.8	2917.6	1865.2	1275.8	**221.3**
U_3_0010_05_2	12.2	386.1	20.3	52.1	*5.7*	13.9	**5.9**	*6.4*	18.3	13.3
U_3_0010_05_5	**11.0**	72.3	24.7	35.6	21.9	25.3	27.9	18.1	12.4	12.8
d25_03_alternative	**336.7**	746.8	510.2	839.4	1675.3	1080.2	1095.6	1192.2	2099.9	837.5
d25_06	36.3	449.8	645.0	**17.4**	76.4	64.3	68.5	69.8	84.8	*30.5*
d25_06_alternative	121.6	237.9	174.5	**50.3**	184.9	144.7	144.7	239.2	137.8	125.9
d25_08	89.6	*37.4*	159.4	**34.5**	219.5	132.1	*64.6*	*37.3*	109.6	*42.5*
gapd_3.min	1838.8	2353.7	**209.9**	>3600.0	3450.6	*1138.6*	>3600.0	*1425.9*	2545.6	*664.3*
p10100-11-115.gq	**85.5**	442.5	119.0	893.7	1828.2	271.1	808.8	669.5	164.0	133.3
p10100-13-105.gq	552.7	>3600.0	*414.1*	1742.9	>3600.0	995.4	1555.4	724.2	952.5	**269.1**
p10100-18-115.gq	363.3	**230.8**	539.2	1299.9	>3600.0	568.0	978.7	710.3	*308.0*	383.9
p1250-10.eq	54.3	277.4	162.4	95.3	139.6	**30.7**	78.0	269.8	54.5	*37.8*
p1250-10.gq	200.5	617.2	250.1	*187.0*	*169.2*	**41.2**	*94.9*	*73.3*	*199.6*	*46.9*
p1250-6.gq	23.5	24.9	*13.3*	50.1	46.4	34.2	36.7	*15.9*	23.9	**12.8**
p1250-7.gq	**57.7**	>3600.0	225.9	113.9	140.0	80.1	66.2	108.1	80.9	77.2
p1650-10.gq	79.1	163.1	238.8	125.6	157.1	*55.5*	*60.3*	164.4	80.6	**42.5**
p2050-10.gq	>3600.0	*484.7*	*86.8*	*31.9*	*33.8*	*23.2*	*40.5*	*484.2*	*18.1*	**14.3**
p2050-8.eq	59.2	115.3	*27.1*	*53.8*	*36.5*	**20.4**	*45.8*	113.6	*24.2*	*21.6*
p2050-8.gq	**17.3**	70.4	19.4	50.6	45.2	19.9	53.9	74.8	26.0	30.6
p23-6.eq	42.2	*32.4*	45.7	45.6	*20.2*	*20.5*	*20.5*	*13.0*	*41.1*	**7.9**
p23-6.gq	25.4	26.4	*21.6*	*18.6*	*20.0*	*14.8*	*16.8*	26.2	*14.6*	**10.5**
p25100-11.gq	353.4	3105.8	*315.3*	536.1	503.8	519.8	*286.4*	685.5	*160.6*	**175.0**
p25100-12.gq	73.6	83.0	138.2	252.5	106.4	*69.5*	*66.5*	**56.4**	216.9	*60.5*
p25100-15.gq	51.9	217.3	**30.9**	250.1	202.9	98.8	140.3	97.1	65.0	78.6
p33100-11.eq	40.0	*25.9*	43.7	219.8	168.3	69.2	148.6	**26.3**	43.1	*32.2*
p33100-11.gq	38.8	**28.1**	39.6	152.4	93.3	47.2	101.6	44.2	*36.1*	43.2
p33100-20.gq	87.3	**52.9**	133.0	190.3	446.2	122.7	246.2	181.7	88.0	139.5
p40100-11-110.eq	67.2	782.5	**63.6**	175.2	158.5	80.9	132.2	775.8	101.0	87.0
p40100-11.eq	>3600.0	>3600.0	>3600.0	*323.2*	*103.3*	**75.2**	*132.0*	*185.9*	*95.6*	*2585.3*
p40100-12-110.eq	18.5	771.4	**15.0**	220.5	411.8	71.8	287.5	773.2	19.3	60.1
p40100-14-115.gq	>3600.0	>3600.0	>3600.0	*123.7*	*174.6*	**64.7**	*141.6*	*70.1*	>3600.0	*537.6*
p40100-15.gq	1413.5	*661.5*	*185.1*	*252.7*	*128.5*	*65.6*	*119.2*	*677.6*	*60.8*	**36.5**
p40100-19.eq	**60.6**	807.8	63.1	719.7	516.5	312.5	509.0	815.1	123.0	84.1
p40100-20.eq	68.4	93.6	71.2	144.7	132.3	78.2	134.6	94.0	69.2	**44.7**
p550-8.gq	732.6	>3600.0	*489.2*	877.4	2703.3	*525.2*	*465.6*	>3600.0	**377.7**	754.5
prob1_050_040_060_005_015_02	219.4	**218.1**	227.0	220.1	220.2	298.9	262.5	220.3	253.5	251.8
prob1_050_040_060_005_015_04	398.6	462.7	464.9	**379.1**	581.9	564.5	558.4	465.7	585.9	560.0
prob1_050_040_060_025_035_10	370.2	**364.6**	370.8	412.2	1346.1	521.9	512.5	522.0	623.5	372.2
prob1_050_090_110_005_015_03	**214.5**	256.4	414.8	856.4	818.1	324.1	321.4	391.5	319.0	321.1
prob1_050_090_110_015_025_01	311.5	**300.4**	334.2	793.3	2554.7	844.5	565.9	*302.6*	742.0	741.9
prob1_050_090_110_025_035_07	228.8	*226.4*	**225.8**	652.8	>3600.0	2761.8	2725.3	*226.7*	2787.7	2792.9
prob1_050_090_110_035_045_07	302.4	*274.9*	**270.2**	326.7	1627.8	476.9	1033.0	406.1	429.9	**289.5**
prob2_050_040_060_015_025_07	225.2	248.4	246.2	374.9	>3600.0	1363.9	1096.5	1566.2	2239.7	**206.4**
prob2_050_040_060_015_025_10	368.0	*232.3*	*233.4*	497.4	>3600.0	1753.2	>3600.0	**232.4**	1729.8	*273.1*
prob2_050_040_060_035_045_03	253.2	426.2	428.1	435.2	>3600.0	1130.1	>3600.0	2002.1	1263.0	**251.5**
prob2_050_040_060_035_045_06	**438.2**	1009.8	1002.4	644.6	>3600.0	2022.9	3403.9	991.3	702.4	514.8
prob2_050_090_110_035_045_06	**277.6**	452.7	638.3	782.5	>3600.0	1137.5	>3600.0	459.9	335.4	412.0
prob3_050_040_060_005_015_01	**260.8**	465.4	458.3	1922.2	>3600.0	3569.5	1293.0	460.8	3342.2	433.6
prob3_050_040_060_015_025_03	322.5	436.7	435.6	813.7	>3600.0	1780.2	2268.6	>3600.0	1703.5	**236.7**
prob3_050_040_060_015_025_09	421.5	500.7	493.1	1160.7	>3600.0	1382.3	>3600.0	496.0	*277.2*	**269.7**
prob3_050_040_060_035_045_10	233.7	236.9	236.6	531.4	>3600.0	877.1	2152.4	239.2	879.9	**212.1**
prob3_050_090_110_005_015_05	239.1	244.4	243.2	1164.9	>3600.0	1195.9	597.3	1882.2	*145.7*	**203.7**
prob3_050_090_110_015_025_01	281.9	**166.8**	**166.8**	540.3	>3600.0	822.1	1195.6	3538.1	*272.0*	299.1
prob3_050_090_110_025_035_10	309.6	475.2	329.3	1072.2	>3600.0	1201.2	1057.8	>3600.0	**252.4**	*254.3*
prob3_050_090_110_035_045_01	153.9	194.0	155.2	669.6	1749.5	*82.5*	472.9	177.4	**81.8**	*143.7*
prob3_050_090_110_035_045_06	445.1	596.1	610.3	1975.9	>3600.0	2200.2	>3600.0	>3600.0	**375.3**	484.1
# best	14	8	8	4	1	8	1	4	6	**22**
# timeouts	5	5	5	2	17	**1**	7	4	3	**1**
geom. mean	149.9	245.1	167.3	248.9	343.0	184.6	246.7	224.5	162.1	**108.4**
arithm. mean	493.8	658.6	500.0	587.8	1251.5	593.8	844.7	685.8	595.1	**335.0**
total (73)	36 049	48 080	36 502	42 912	91 357	43 347	61 660	50 062	43 439	**24 452**

The following table gives the number of nodes needed to solve each problem instance with original variable branching. The best entry in each row except full strong entries is in **bold face**.

	pseudo-cost	random	most-frac	SBw/oCG	SBw/CG	hierar-chical	hybrid	reli-able	rel. hier.	hybrid hier.
12Cap10	1 009	7 083	2 234	351	137	186	143	7 083	**137**	280
14Cap10	643	422	1 056	102	52	68	**47**	422	643	128
20Cap10	795	2 575	1 569	318	112	116	**95**	2 575	795	135
NU_1_0010_05_3	4 672	4 439	4 672	4 352	4 397	4 397	4 397	4 397	4 439	**4 284**
NU_1_0010_05_7	**2 495**	4 576	**2 495**	3 713	2 643	12 338	2 652	2 643	3 730	3 278
NU_3_0010_05_1	1 368	4 229	1 490	1 958	415	**566**	800	4 229	920	2 253
NU_3_0010_05_3	3 135	9 291	3 261	7 696	1 293	3 437	3 082	9 291	3 061	**2 581**
NU_3_0010_05_5	2 607	4 055	2 488	2 477	132	897	288	**174**	837	1 441
NU_3_0010_05_7	1 207	2 591	1 479	979	376	859	**711**	2 591	1 375	1 389
NU_3_0010_05_9	1 993	3 444	2 669	2 432	544	1 862	1657	3 444	2 234	**1 473**
U_1_0050_05_0	**325**	**325**	**325**	**325**	325	**325**	**325**	**325**	**325**	**325**
U_1_0050_05_2	8 838	17 396	4 441	36 523	10 365	9 005	6 931	17 396	**1 085**	19 868
U_1_0050_25_7	>258 966	4 897	>264 048	>14 046	1 062	2 268	4 367	4 897	>194 963	**402**
U_1_0100_05_1	19 639	12 568	26 890	12 563	12 563	9 957	6 617	12 563	9 126	**661**
U_1_0100_05_3	**129 326**	**129 326**	**129 326**	**129 326**	129 326	**129 326**	**129 326**	**129 326**	**129 326**	**129 326**
U_2_0050_05_4	32 297	4 384	17 003	14 893	5 907	12 635	110 850	**3 518**	10 202	7 620
U_2_0100_05_2	>109 482	25 367	>116 151	**24 166**	>90 663	>103 350	>113 381	25 367	>141 143	>109 421
U_2_0100_05_5	49 779	28 075	>176 504	**15 139**	9 910	66 956	117 682	5 1078	48 038	15 838
U_3_0010_05_2	2 537	84 845	4 749	8 435	225	2 675	358	**244**	3 925	2 941
U_3_0010_05_5	1 876	8 944	3 961	4 882	1 256	3 853	3 832	**1 100**	1 996	2 961
d25_03_alternative	21 260	46 586	31 697	19 854	12 852	**9 287**	26 475	10 834	18 637	49 201
d25_06	795	9 663	11 073	194	168	**156**	169	206	192	618
d25_06_alternative	4 451	8 289	6 529	1 409	1 419	**1 352**	1 557	8 289	5 075	4 982
d25_08	1 891	824	4 016	222	248	234	737	824	**208**	862
gapd_3.min	10 461	9 512	1 629	>484	440	**595**	>580	798	927	2 886
p10100-11-115.gq	855	2 566	1 088	196	151	**138**	175	522	1 664	519
p10100-13-105.gq	1 635	>6 212	2 020	168	>67	271	**108**	318	253	1 112
p10100-18-115.gq	1 194	984	1 440	299	>158	271	389	266	**195**	1 205
p1250-10.eq	2 012	4 720	2 844	331	211	**195**	287	4 720	2 012	1 083
p1250-10.gq	3 919	5 686	2 412	331	171	**174**	271	373	3 919	783
p1250-6.gq	1 067	1 359	659	171	95	**143**	209	151	1 067	336
p1250-7.gq	1 749	>15 329	2 620	480	332	**424**	1 277	799	465	1 426
p1650-10.gq	2 907	4 648	3 138	539	420	**304**	411	4 648	2 907	1 636
p2050-10.gq	>10 266	4 988	1 983	136	69	97	**86**	4 988	87	160
p2050-8.eq	1 376	2 840	950	299	139	**144**	154	2 840	180	584
p2050-8.gq	826	2 658	960	269	135	**112**	161	2 658	146	817
p23-6.eq	1 195	1 345	1 279	348	123	187	259	**147**	1 313	264
p23-6.gq	1 266	1 321	1 079	173	129	161	175	1 321	**131**	307
p25100-11.gq	19 308	145 400	18 307	988	506	1 125	1 685	2 329	**448**	7 212
p25100-12.gq	4 680	5 131	8 632	543	121	174	**146**	206	11 938	1 376
p25100-15.gq	867	2 219	438	299	122	183	162	284	**140**	575
p33100-11.eq	2 046	1 146	1 939	314	143	**103**	159	1 146	2 171	339
p33100-11.gq	1 811	1 282	1 973	252	73	**106**	118	117	1 785	492
p33100-20.gq	1 483	1 116	1 022	326	151	**140**	518	527	1 483	2 490
p40100-11-110.eq	2 365	7 009	2 259	171	132	165	**98**	7 009	181	1 459
p40100-11.eq	>18 707	>18 357	>16 927	399	80	**107**	128	613	154	7 240
p40100-12-110.eq	641	6 192	463	154	133	**90**	119	6 192	659	263
p40100-14-115.gq	>6 929	>8 678	>6 994	144	112	**121**	152	221	>6 950	2 790
p40100-15.gq	3 117	3 923	1 641	275	73	129	141	3 923	**87**	262
p40100-19.eq	3 049	38 708	3 184	603	265	333	5 858	38 708	**191**	1 597
p40100-20.eq	1 538	1 804	1 244	158	101	122	**105**	1 804	1 538	270
p550-8.gq	2 183	>6 661	1 985	507	439	**449**	549	>6 648	485	2 706
prob1_050_040_060_005_015_02	**1 049**	**1 049**	**1 049**	**1 049**	1 049	1 154	1 154	**1 049**	1 154	1 154
prob1_050_040_060_005_015_04	1 804	2 204	2 204	**1 793**	1 823	2 337	2337	2 204	2 337	2 337
prob1_050_040_060_025_035_10	1 716	1 712	1 712	1 712	1 728	1 853	1 780	**1 687**	1 870	1 726
prob1_050_090_110_005_015_03	**1 047**	1 183	2 150	2 240	1 616	1 400	1 400	1 610	1 400	1 400
prob1_050_090_110_015_025_01	1 626	1 928	1 928	1 928	>432	1 310	**684**	1 928	1 266	1 638
prob1_050_090_110_025_035_07	1 668	1 668	1 668	**1 116**	>584	2 594	2 646	1 668	2 594	2 594
prob1_050_090_110_035_045_07	1 265	1 247	1 247	1 249	1 253	1 483	1 420	**1 230**	1 474	1 255
prob2_050_040_060_015_025_07	1 276	1 517	1 517	1 516	>603	3 418	**935**	1 510	6 275	1 117
prob2_050_040_060_015_025_10	1 657	**1 066**	**1 066**	1 479	>168	1 846	>375	**1 066**	1 755	1 162
prob2_050_040_060_035_045_03	1 171	1 809	1 809	1 429	>181	1 298	>721	1 031	1 233	**1 120**
prob2_050_040_060_035_045_06	2 018	4 957	4 933	2 294	>352	2 976	**751**	4 957	2 520	2 245
prob2_050_090_110_035_045_06	1 145	2 021	2 711	1 466	>161	1 163	>382	2 021	**1 116**	1 602
prob3_050_040_060_005_015_01	2 151	4 027	3 948	3 817	>694	4 348	**496**	4 027	3 980	3 636
prob3_050_040_060_015_025_03	2 199	3 606	3 606	3 628	>544	2 864	**869**	>2 887	2 655	1 626
prob3_050_040_060_015_025_09	2 794	3 931	3 931	3 918	>311	1 963	>594	3 931	**1 890**	1 923
prob3_050_040_060_035_045_10	1 240	1 280	1 423	1 280	1 320	1 543	**1 087**	1 280	1 549	1 543
prob3_050_090_110_005_015_05	1 387	1 789	1 789	1 789	>162	1 539	**90**	1 627	997	1 435
prob3_050_090_110_015_025_01	1 730	993	993	1 035	>102	**842**	**171**	995	1 641	1 834
prob3_050_090_110_025_035_10	2 053	3 729	2 563	2 561	>264	1 723	**180**	>2 823	1 855	1 865
prob3_050_090_110_035_045_01	1 119	1 691	1 198	1 116	129	95	**72**	129	93	1 087
prob3_050_090_110_035_045_06	2 996	4 413	4 413	4 412	>169	2 873	>613	>2 637	**2 499**	3 293
# best (w/o SBw/CG)	5	4	5	7	-	**25**	18	11	14	11
geom. mean	2 689.8	4 020.1	3 010.5	1 089.2	418.7	799.3	**674.2**	1 801.1	1 413.3	1 507.0
arithm. mean	11 010.5	10 645.8	13 111.1	**4 904.5**	4 173.8	5 792.3	7 804.3	6 019.0	9 151.2	6 018.4
total (73)	803 766	777 147	957 108	**358 032**	304 687	422 841	569 716	439 389	668 041	439 343