Bestuzheva, Ksenia et al.

Article — Published Version

Global optimization of mixed-integer nonlinear programs with SCIP 8

Journal of Global Optimization

Provided in Cooperation with:
Springer Nature

Suggested Citation: Bestuzheva, Ksenia et al. (2023) : Global optimization of mixed-integer nonlinear programs with SCIP 8, Journal of Global Optimization, ISSN 1573-2916, Springer US, New York, NY, pp. 1-24, https://doi.org/10.1007/s10898-023-01345-1

This Version is available at:
https://hdl.handle.net/10419/308744

Journal of Global Optimization
https://doi.org/10.1007/s10898-023-01345-1

Global optimization of mixed-integer nonlinear programs with SCIP 8

Ksenia Bestuzheva[1] · **Antonia Chmiela**[1] · **Benjamin Müller**[1] · **Felipe Serrano**[1] · **Stefan Vigerske**[2] · **Fabian Wegscheider**[1]

Received: 16 May 2023 / Accepted: 12 November 2023

Abstract
For over 10 years, the constraint integer programming framework SCIP has been extended by capabilities for the solution of convex and nonconvex mixed-integer nonlinear programs (MINLPs). With the recently published version 8.0, these capabilities have been largely reworked and extended. This paper discusses the motivations for recent changes and provides an overview of features that are particular to MINLP solving in SCIP. Further, difficulties in benchmarking global MINLP solvers are discussed and a comparison with several state-of-the-art global MINLP solvers is provided.

Keywords Global optimization · Mixed-integer nonlinear programming · SCIP · Branch-and-cut · Optimization software · Benchmark

Mathematics Subject Classification 65K05 · 90-08 · 90C11 · 90C20 · 90C26

1 Introduction

Mixed-integer nonlinear programming (MINLP) concerns with the optimization of an objective function such that a finite set of linear or nonlinear constraints and integrality conditions is satisfied. The generality of this problem class means that many real-world applications can

✉ Stefan Vigerske
svigerske@gams.com

Ksenia Bestuzheva
bestuzheva@zib.de

Antonia Chmiela
chmiela@zib.de

Benjamin Müller
benjamin.mueller@zib.de

Felipe Serrano
serrano@zib.de

[1] Department AIS2T, Zuse Institute Berlin, Berlin, Germany

[2] GAMS Software GmbH, c/o Zuse Institute Berlin, Berlin, Germany

be modeled as MINLP [1–4], but also that software that handles this class efficiently becomes extremely complex. MINLP solvers [5] are often built on top of or by combining solvers for mixed-integer linear programs (MIP) and solvers that find locally optimal solutions for nonlinear programs (NLP). In fact, the first general purpose solver, DICOPT [6], decomposes the solution of an MINLP into a sequence of MIP and NLP solves [7], thereby building on established software for these program classes. DICOPT solves MINLPs with convex nonlinear constraints to optimality, but works only as a heuristic on nonconvex MINLPs. The first general purpose solvers for nonconvex MINLPs were αBB, BARON, and GLOP [8–10], all based on convexification techniques for nonconvex constraints. Also the solver SCIP (Solving Constraint Integer Programs) belongs to this category [11].

In the following, MINLPs of the form

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

are considered, where $\underline{x}, \overline{x} \in \overline{\mathbb{R}}^n$, $\overline{\mathbb{R}} := \mathbb{R} \cup \{\pm\infty\}$, $\underline{x} \le \overline{x}$, $\mathcal{I} \subseteq \{1, \ldots, n\}$, $c \in \mathbb{R}^n$, $\underline{g}$, $\overline{g} \in \overline{\mathbb{R}}^m$, $\underline{g} \le \overline{g}$, $g : \mathbb{R}^n \to \overline{\mathbb{R}}^m$ is specified explicitly in algebraic form, $\underline{b}, \overline{b} \in \overline{\mathbb{R}}^{\tilde{m}}$, $\underline{b} \le \overline{b}$, and $A \in \mathbb{R}^{\tilde{m} \times n}$. The restriction to a linear objective function is a technical detail of SCIP and without loss of generality.

SCIP is a branch-cut-and-price framework for the solution different types of optimization problems, most generally *constraint integer programs* (CIPs), and most importantly MIPs and MINLPs. 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 is a linear or nonlinear program. The problem class of CIP was motivated by the modeling flexibility of constraint programming and the algorithmic requirements of integrating it with efficient solution techniques for MIP [12].

In order to solve CIPs, SCIP constructs relaxations—typically linear programs (LPs). If the relaxation solution is not feasible for the current subproblem, the plugins that handle the violated constraints need to take measures to eventually render the relaxation solution infeasible for the updated relaxation, for example by branching or separation [12]. A plethora of additional plugin types, e.g., for presolving, finding feasible solutions, or tightening variable bounds, allow accelerating the solution process. After 20 years of development of the framework itself and included plugins, SCIP includes mature solvers for MIP, MINLP, and several other problem classes [13]. The extended version of this paper [14] provides a short overview on the history of the MINLP solver in SCIP. Since November 2022, SCIP is freely available under an open-source license.

SCIP solves MINLPs to global optimality via a spatial branch-and-bound algorithm that mixes branch-and-infer and branch-and-cut [15]. Important parts of the solution algorithm are presolving, domain propagation (that is, tightening of variable bounds), linear relaxation, and branching. A distinguishing feature of SCIP is that its capabilities to handle nonlinear constraints are not limited to MINLPs, but can be used for any CIP. For example, problems can be handled where linear and nonlinear constraints are mixed with typical constraints from constraint programming, as long as appropriate constraint handlers have been included in SCIP. Since most constraint handlers in SCIP construct a linear relaxation of their constraints,

also the handling of nonlinear constraints focuses on linear relaxations. The emphasis on handling CIPs with nonlinear constraints rather than MINLP only is also a reason that the use of nonlinear relaxations or reformulations of complete MINLPs into other problem types, e.g., mixed-integer conic programs, has not been explored much so far.

With SCIP 8 [16], a complete overhaul of nonlinear constraint handling was released. The primary motivation for this change was to increase the reliability of the solver and to alleviate numerical issues that arose from problem reformulations and led to SCIP returning solutions that are feasible in the reformulated problem, but infeasible in the original problem. More precisely, previous SCIP versions built an extended formulation of (MINLP) explicitly, with the consequence that the original constraints were no longer included in the presolved problem. Even though the formulations were theoretically equivalent, it was possible that ε-feasible solutions for the reformulated problem were not ε-feasible in the original problem. SCIP 8 remedies this by building an implicit extended formulation as an annotation to the original problem. A second motivation for the major changes in SCIP 8 was to reduce the ambiguity of expression and nonlinear structure types by implementing different plugin types for low-level structure types that define expressions, and high-level structure types that add functionality for particular, sometimes overlapping structures. Finally, new features for improving the solver's performance on MINLPs were introduced. These include intersection, SDP (semi-definite programming), and RLT (reformulation linearization technique) cuts for quadratic expressions [17, 18], perspective strengthening [19], and symmetry detection [20].

An overview of SCIP's MINLP solving capabilities is given next. Afterwards, the performance of SCIP and other global MINLP solvers is compared.

2 MINLP capabilities of SCIP

In the following, the integration of nonlinear constraints into the branch-and-cut solver of SCIP is discussed. Next, the concept of a *nonlinear handler* is introduced, which is a new plug-in type of SCIP 8 that facilitates the integration of extensions that handle specific nonlinear structures. The remainder of this section gives a concise overview of features that increase the efficiency of MINLP solving. Unless specified otherwise, more details are often found in [16].

2.1 Framework

2.1.1 Expressions

Algebraic expressions are well-formed combinations of constants, variables, and algebraic operations such as addition, multiplication, and exponentiation, that are used to describe mathematical functions. They are represented by a directed acyclic graph with nodes representing variables, constants, and operations and arcs indicating the flow of computation. In SCIP, all semantics of expression operands are defined by *expression handler* plugins. These handler provide callbacks that are used by the SCIP core to manage expressions (create, modify, copy, parse, print), to evaluate at a point or over intervals, to compute derivatives, to simplify and compare, and to check curvature and integrality.

For the following operators, expression handlers are included in SCIP 8: constant, variable, affine-linear function, product, power, signpower ($y \mapsto \operatorname{sign}(y)|y|^p$ for $p > 1$), exponentiation, logarithm, entropy, sine, cosine, and absolute value. In previous versions of SCIP, also

high-level structures such as quadratic functions could be represented as expression types. To avoid ambiguity and reduce complexity, this has been replaced by a recognition of quadratic expressions that is no longer made explicit in the expression type.

2.1.2 Constraint handler for nonlinear constraints

All nonlinear constraints $\underline{g} \leq g(x) \leq \overline{g}$ of (MINLP) are handled by the constraint handler for nonlinear constraints in SCIP, while the linear constraints $\underline{b} \leq Ax \leq \overline{b}$ are handled by the constraint handlers for linear constraints and its specializations (e.g., knapsack, set-covering). A constraint handler is responsible for checking whether solutions satisfy constraints and, if that is not the case, to resolve infeasibility by *enforcing constraints*. This applies in particular to solutions of the LP relaxation. The nonlinear constraint handler currently enforces its constraints by the following means:

DOMAINPROP: by analyzing the constraints with respect to the variable bounds at the current node of the branch-and-bound tree, infeasibility or a bound tightening may be deduced, which allow pruning the node or cutting off the given solution, respectively; this is also known as *domain propagation*;
SEPARATE: computing a cutting plane that is violated by the given solution;
BRANCH: the current node of the branch-and-bound tree is subdivided, that is, a variable x_i and a branching point $\tilde{x}_i \in [\underline{x}_i, \overline{x}_i]$ are selected and two child nodes with x_i restricted to $[\underline{x}_i, \tilde{x}_i]$ and $[\tilde{x}_i, \overline{x}_i]$, respectively, are created.

To decide whether a node can be pruned (DOMAINPROP), an overestimate of the range of $g(x)$ with respect to current variable bounds is computed by means of interval arithmetics [21]. If a constraint k is found such that $g_k([\underline{x}, \overline{x}]) \cap [\underline{g}_k, \overline{g}_k] = \emptyset$, then there exists no point in $[\underline{x}, \overline{x}]$ for which this constraint is feasible. A bound tightening may be computed by applying the same methods in reverse order. That is, interval arithmetic is used to overestimate $g^{-1}([\underline{g}, \overline{g}])$, the preimage of $g(x)$ on $[\underline{g}, \overline{g}]$, and variable bounds are tightened to $[\underline{x}, \overline{x}] \cap g^{-1}([\underline{g}, \overline{g}])$. This is also known as feasibility-based bound tightening (FBBT). In the simplest case, callbacks of expression handlers are used to propagate intervals through expressions. However, in some cases, other methods that take more structure into account or that use additional information are used (see, e.g., Sects. 2.3.1 and 2.3.2).

To construct a linear relaxation of the nonlinear constraints (SEPARATE option), an extended formulation is considered:

$$
\begin{aligned}
&\min\ c^\top x, && && (\text{MINLP}_{\text{ext}})\\
&\text{such that } h_i(x, w_{i+1}, \ldots, w_{\hat{m}}) \lesseqgtr_i w_i, && i = 1, \ldots, \hat{m},\\
&\qquad \underline{b} \leq Ax \leq \overline{b},\\
&\qquad \underline{x} \leq x \leq \overline{x}, \quad \underline{w} \leq w \leq \overline{w},\\
&\qquad x_{\mathcal{I}} \in \mathbb{Z}^{|\mathcal{I}|}.
\end{aligned}
$$

The functions h_i are obtained from the expressions that define functions g_i by recursively annotating subexpressions with auxiliary variables $w_{i+1}, \ldots, w_{\hat{m}}$ for some $\hat{m} \geq m$. Initially, slack variables $w_1, \ldots, w_m$ are introduced and assigned to the root of all expressions, i.e., $h_i := g_i, \underline{w}_i := \underline{g}_i, \overline{w}_i := \overline{g}_i$, for $i = 1, \ldots, m$. Next, for each function h_i, subexpressions f may be assigned new auxiliary variables $w_{i'}, i' > m$, which results in extending ($\text{MINLP}_{\text{ext}}$) by additional constraints $h_{i'}(x) = w_{i'}$ with $h_{i'} := f$. Bounds $\underline{w}_{i'}$ and $\overline{w}_{i'}$ are initialized to bounds on $h_{i'}$, if available. Since auxiliary variables in a subexpression of h_i always receive

an index larger than $\max(m, i)$, the result is referred to by $h_i(x, w_{i+1}, \ldots, w_{\hat{m}})$ for any $i = 1, \ldots, \hat{m}$. If a subexpression appears in several expressions, it is assigned at most one auxiliary variable.

For the (in)equality sense $\lesseqgtr_i$, a valid simplification is to assume equality. For performance reasons, though, it can be beneficial to relax to inequalities if that does not change the feasible space of (MINLP$_{\text{ext}}$) when projected onto x. Therefore, for $i \in \{1, \ldots, m\}$, $\lesseqgtr_i$ is set according to the finiteness of $\underline{g}_i$ and $\overline{g}_i$. For $i > m$, monotonicity of expressions is taken into account to derive $\lesseqgtr_i$.

Whether to annotate a subexpression by an auxiliary variable depends on the structures that are recognized. In the simplest case, every subexpression that is not already a variable is annotated with an auxiliary variable. This essentially corresponds to the Smith Normal Form [10]. For every function h_i of (MINLP$_{\text{ext}}$), the callbacks of the corresponding expression handler can be used to compute linear under- and overestimators, such that a linear relaxation for (MINLP$_{\text{ext}}$) is constructed. It can, however, be beneficial to not add an auxiliary variable for every subexpression, thus allowing for more complex functions in (MINLP$_{\text{ext}}$). This will be the discussed in Sect. 2.1.3 below.

If a constraint $h_i(x, w_{i+1}, \ldots, w_{\hat{m}}) \lesseqgtr_i w_i$ of (MINLP$_{\text{ext}}$) is violated in the LP solution and no cut is found that separates this solution, then the variables appearing in h_i are candidates for branching (BRANCH). More precisely, when an expression handler computes a linear under- or overestimator for $h_i(x, w_{i+1}, \ldots, w_{\hat{m}})$, it also signals for which variables it used current variable bounds. Marked original variables are then added to the list of branching candidates. For an auxiliary variable $w_{i'}$, $i' > i$, the variables in the subexpression that $h_{i'}$ represents are considered for branching instead.

The decision on whether to add a cutting plane that separates the solution of the LP relaxation or to branch is rather complex, but the idea is to branch if either no cutting plane is found or if the violation of available cutting planes in the relaxation solution is rather small when compared to the convexification gap of the under/overestimators that define the cutting planes. In the latter case, it may be beneficial to first reduce the convexification gap by branching. To select one variable from the list of branching candidates, the violation of constraints in (MINLP$_{\text{ext}}$) and historical information about the effect of branching on a given variable on the optimal value of the LP relaxation ("pseudo costs") are taken into account. The branching point is a convex combination of the value of the variable in the LP relaxation and the mid-point of the variable's interval.

2.1.3 Nonlinear handlers

For a constraint $\log(x)^2 + 2\log(x)y + y^2 \leq 4$, a slack variable and four auxiliary variables would be introduced to construct the extended formulation $w_2 + 2w_3 + w_4 \leq w_1$, $w_5 = \log(x)$, $w_2 = w_5^2$, $w_3 = w_5 y$, $w_4 = y^2$. This is due to the expression handlers having a rather myopic view, basically, implementing techniques that can handle only their direct children. It is clear that, for this example, an extended formulation that only replaces $\log(x)$ by an auxiliary variable w_2 could be more efficient to solve. However, this requires methods to detect the quadratic (or convex) structure and to either compute linear underestimators for the quadratic (convex) expression $w_2^2 + 2w_2 y + y^2$ or to separate cutting planes for the set defined by $w_2^2 + 2w_2 y + y^2 \leq w_1$.

Such structure detection and handling methods are the task of the new *nonlinear handler* plugins that were introduced with SCIP 8. Nonlinear handlers determine the extended formulation (MINLP$_{\text{ext}}$) by deciding when to annotate subexpressions with auxiliary variables. That

is, given a constraint $h_i(x) \lesseqgtr_i w_i$, a nonlinear handler analyses the expression that defines h_i and attempts to detect specific structures. At this point, it may also request to introduce additional auxiliary variables, thus changing $h_i(x)$ into $h_i(x, w_{i+1}, \ldots, w_{\hat{m}})$. In addition, it informs the constraint handler that it will provide separation for $h_i(x, w_{i+1}, \ldots, w_{\hat{m}}) \leq w_i$, or $\geq w_i$, or both. If none of the nonlinear handlers declare that they will handle $h_i(x) \lesseqgtr_i w_i$, auxiliary variables are introduced for each argument of the root of the expression h_i and expression handler callbacks are used to construct cutting planes from linear under-/overestimators.

In addition to separation, nonlinear handlers can also contribute to domain propagation. This is implemented analogously to separation by setting up an additional extended formulation similarly to ($\text{MINLP}_{\text{ext}}$).

Note that the extended formulations are stored as *annotation* on the original expressions. Thus, for each task, the most suitable formulation can be used. For example, feasibility is checked on the original constraints, domain propagation and separation use the corresponding extended formulations, but branching is performed, by default, with respect to original variables only. With SCIP 7 and earlier, only one extended formulation was constructed explicitly and the connection to the original formulation was no longer available, leading to problems in ensuring that solutions are (ε-)feasible for the original constraints.

In addition to an improved numeric reliability, the nonlinear handlers also allow for a higher flexibility when handling nonlinear structures. For each node in an expression, several nonlinear handler can be attached, each one annotating possibly different subexpressions with auxiliary variables. For example, for a nonconvex quadratic constraint $\sum_{i,j} a_{i,j} x_i x_j \leq w$, the nonlinear handler for quadratics can declare that it will provide separation (by intersection cuts, see Sect. 2.3.5), but that also other means of separation should be tried. However, since no other nonlinear handler declares that it will provide separation, auxiliary variables are introduced for each argument of the sum, that is, an auxiliary variable X_{ij} is assigned to each product $x_i x_j$. For the corresponding constraints $x_i x_j \leq X_{ij}$ (if $a_{i,j} \geq 0$), the McCormick underestimators [22]

$$X_{ij} \geq \underline{x}_i x_j + \underline{x}_j x_i - \underline{x}_i \underline{x}_j, \quad X_{ij} \geq \overline{x}_i x_j + \overline{x}_j x_i - \overline{x}_i \overline{x}_j \tag{1}$$

or other means (see Sect. 2.3.2) will be used to construct a linear relaxation.

2.1.4 NLP relaxation

Similar to the central LP relaxation of SCIP, an NLP relaxation is also available. In contrast to constraint handlers, the NLP relaxation uses a common data structure to store its constraints. Therefore, in case of a MINLP, the NLP relaxation together with the integrality conditions on variables provides a unified view of the problem. To find local optimal solutions for the NLP relaxation, interfaces to the NLP solvers filterSQP, Ipopt, and Worhp [23–25] are available. Function derivatives are computed via CppAD [26].

2.2 Presolving

When presolving nonlinear constraints, expressions are simplified and brought 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. In addition, it is ensured that if a subexpression appears several times (in the same or different constraints), always the same expression object is used.

2.2.1 Variable fixings

Similar to what has been shown by Hansen et al. [27], if a bounded variable x_j does not appear in the objective ($c_j = 0$), but in exactly one constraint $\underline{g}_k \leq g_k(x) \leq \overline{g}_k$ where $g_k(x)$ is convex in x_j for any fixing of other variables and $\overline{g}_k = +\infty$ (or concave in x_j and $\underline{g}_k = -\infty$), then there always exists an optimal solution where $x_j \in \{\underline{x}_j, \overline{x}_j\}$. For example, if $y \in [0, 1]$ appears only in a constraint $xy + yz - y^2 \leq 5$, then y can be changed to a binary variable.

SCIP recognizes such variables for polynomial constraints (under additional assumptions [16]) and changes the variable type to binary, if $\underline{x}_j = 0$ and $\overline{x}_j = 1$, or adds a bound disjunction constraint $x_j \leq \underline{x}_j \vee x_j \geq \overline{x}_j$. As a consequence, branching on x_j leads to fixing the variable in both children.

2.2.2 Linearization of products

To better utilize SCIP's techniques for MIP solving, products of binary variables are linearized. In the simplest case, a product $\prod_i x_i$ is replaced by a new variable z and a constraint of type "and" that models $z = \bigwedge_i x_i$ is added. The "and"-constraint handler will then separate a linearization of this product [28]. For a product of only two binary variables, the linearization is added directly.

For a quadratic function in binary variables with many terms, the number of variables introduced may be large. In this case, a linearization that requires fewer additional variables is used, even though it may lead to a weaker relaxation.

2.2.3 KKT strengthening for QPs

A presolving method that aims to tighten the relaxation of a quadratic program (QP) by adding redundant constraints derived from Karush-Kuhn-Tucker (KKT) conditions is available. Consider a quadratic program of the form $\min\{\frac{1}{2} x^\top Q x + c^\top x : Ax \leq b\}$, where $Q \in \mathbb{R}^{n \times n}$ is symmetric, $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, and $b \in \mathbb{R}^m$. If the QP is bounded, then all optima satisfy the KKT conditions $Qx + c + A^\top \mu = 0$, $Ax \leq b$, $\mu_i(Ax - b)_i = 0$, $i = 1, \ldots, m$, where $\mu \geq 0$ is the vector of Lagrangian multipliers of constraints $Ax \leq b$.

If SCIP recognizes that (MINLP) is equivalent to a QP and all variables are bounded, then the KKT conditions are added as redundant constraints to the problem, whereby the complementarity constraints are formulated via special ordered sets of type 1. The redundant constraints can help to strengthen the linear relaxation and prioritize branching decisions to satisfy the complementarity constraints, which focuses the search more on the local optima.

In addition to a QP, the implementation can also handle mixed-binary quadratic programs. For all details, see [29, 30]. When this presolver was added to SCIP 4.0, it has shown to be very beneficial for box-constrained quadratic programs. Due to the many changes and extensions in SCIP 8 for the handling of quadratic constraints (Sect. 2.3) it needs to be reevaluated under which conditions this presolver should be enabled. Currently, it is disabled by default.

2.2.4 Symmetry detection

Symmetries are automorphisms on $\mathbb{R}^n$ that map optimal solutions to optimal solutions. They have an adverse effect on the performance of branch-and-bound solvers, because symmetric subproblems may be treated repeatedly. Therefore, SCIP can enforce lexicographically

maximal solutions from an orbit of symmetric solutions via bound tightening and separation [16, 31–33].

Since optimal solutions are naturally not known in advance, the symmetry detection resorts to find permutations of variables that map the feasible set onto itself and map each point to one with the same objective function value [34]. These permutations are given by isomorphisms in an auxiliary symmetry detection graph, which is constructed from the problem data (e.g., c, A, $\mathcal{I}$, and the expressions that define $g(x)$) [20, 35].

2.3 Quadratics

Since quadratic functions frequently appear in MINLPs, a number of techniques have been added to SCIP to handle this structure. Next to the presolving methods that were discussed in the previous section, three nonlinear handlers and four separators deal with quadratic structures. When none of the nonlinear handlers are active, then for each square and bilinear term in a quadratic function, an auxiliary variable is added in the extended formulation and gradient, secant, and McCormick under- and overestimators (1) are generated.

2.3.1 Domain propagation

If variables appear more than once in a quadratic function, then a term-wise domain propagation does not necessarily yield the best possible results, due to suffering from the so-called *dependency problem* of interval arithmetics. For example, it is easy to compute the range for $x^2 + x$ for given bounds on x, or bounds on x for a given interval on $x^2 + x$, but standard interval arithmetics treats the terms x^2 and x separately, which leads to overestimating the result.

Therefore, a specialized nonlinear handler in SCIP provides a domain propagation procedure for quadratics that aims to reduce overestimation. For this, the detection routine of the handler writes a quadratic expression as $q(y) = \sum_{i=1}^{k} q_i(y)$ with $q_i(y) = a_i y_i^2 + c_i y_i + \sum_{j \in P_i} b_{i,j} y_i y_j$, where y_i is either an original variable (x) or another expression, $a_i, c_i \in \mathbb{R}$, $b_{i,j} \in \mathbb{R}\backslash\{0\}$, $j \in P_i \Rightarrow i \notin P_j$ for all $j \in P_i$, $P_i \subset \{1, \ldots, k\}$, $i = 1, \ldots, k$. For functions q_i with at least two terms (at least two of $a_i, b_{i,j}, j \in P_i$, and c_i are nonzero), a relaxation is obtained by replacing each y_j by $[\underline{y}_j, \overline{y}_j]$, $j \in P_i$. For this univariate quadratic interval-term in y_i, tight bounds can be computed [36].

In addition, bounds on variables y_j, $j \in P_i$, are computed by considering $\sum_{j \in P_i} b_{i,j} y_j \in ([\underline{q}, \overline{q}] - \sum_{i' \neq i} q_{i'}(y))/y_i - a_i y_i - c_i$, $y_i \in [\underline{y}_i, \overline{y}_i]$, where $[\underline{q}, \overline{q}]$ are given bounds on $q(y)$. After relaxing each $q_{i'}$ to an interval, bounds on each y_j, $j \in P_i$, can be computed.

2.3.2 Bilinear terms

For a product $y_1 y_2$, where y_1 and y_2 are either non-binary variables or other expressions, best possible linear under- and overestimators when considering the bounds $[\underline{y}_1, \overline{y}_1] \times [\underline{y}_2, \overline{y}_2]$ only are given by (1). However, if linear inequalities in y_1 and y_2 are available, then possibly tighter linear estimates and variable bounds can be computed using an algorithm by Locatelli [37]. The inequalities are found by projection of the LP relaxation onto variables (y_1, y_2). For more details, see [38]. An alternative method that uses linear constraints to tighten the relaxation of quadratic constraints is described in the following.

2.3.3 RLT cuts

The Reformulation–Linearization Technique (RLT) [39, 40] has proven very useful to tighten relaxations of polynomial programming problems. In SCIP, an RLT separator for bilinear product relations in ($\text{MINLP}_{\text{ext}}$) is available.

For simplicity, denote by X_{ij} the auxiliary variable that is associated with a constraint $x_i x_j \lesseqgtr X_{ij}$ of ($\text{MINLP}_{\text{ext}}$) ($X_{ji}$ denotes the same variable as X_{ij}). Recall that it is valid to replace $\lesseqgtr$ by $=$. RLT cuts are derived by multiplying a linear constraint by a nonnegative bound factor and replacing the product relations by variables from X. For example, given a linear constraint $a^\top x \leq b$ and a bound $x_i \geq \underline{x}_i$, the quadratic inequality $a^\top x\,(x_i - \underline{x}_i) \leq b\,(x_i - \underline{x}_i)$ is formed. Next, each term $x_k x_i$ is replaced by X_{ki}, if $X_{ki} = x_k x_i$ exists in ($\text{MINLP}_{\text{ext}}$), or estimated by (1), otherwise.

In addition, the RLT separator can reveal linearized products between binary and continuous variables. To do so, it checks whether pairs of linear inequalities that are defined in the same triple of variables (one of them binary, the other two continuous) imply a product relation. These implicit products can then be used in the linearization step of RLT cut generation [18].

2.3.4 SDP cuts

A popular convex relaxation of the condition $X = xx^\top$ (see previous section) is given by requiring $X - xx^\top$ to be positive semidefinite (psd). 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. Therefore, only principal 2×2 minors of $X - xx^\top$, which also need to be psd, are considered. By Schur's complement, this means that the condition

$$A_{ij}(x, X) := \begin{bmatrix} 1 & x_i & x_j \\ x_i & X_{ii} & X_{ij} \\ x_j & X_{ij} & X_{jj} \end{bmatrix} \succeq 0 \tag{2}$$

needs to hold for any i, j, $i \neq j$. A separator in SCIP detects minors for which X_{ii}, X_{jj}, X_{ij} exist in ($\text{MINLP}_{\text{ext}}$) and enforces $A_{ij}(x, X) \succeq 0$ by adding a linear inequality $v^\top A_{ij}(x, X)v \geq 0$, where $v \in \mathbb{R}^3$ is an eigenvector of $A_{ij}(\hat{x}, \hat{X})$ with $v^\top A_{ij}(\hat{x}, \hat{X})v < 0$ and $(\hat{x}, \hat{X})$ is the solution that violates (2).

2.3.5 Intersection cuts

Intersection cuts [41, 42] have shown to be efficient to strengthen relaxations of MIPs. A recently described method to compute the tightest possible intersection cuts for quadratic programs [43] has been implemented in SCIP [17].

Assume a nonconvex quadratic constraint of ($\text{MINLP}_{\text{ext}}$) is $q(y) \leq w$ with q being a quadratic as in Sect. 2.3.1. The separation of intersection cuts is implemented 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, but using *maximal* S-free sets yield the tightest possible intersection cuts [43].

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. To obtain even better cuts, there is also a strengthening procedure implemented that uses the idea of negative edge extension of the cone K [44].

In addition to the separation of intersection cuts for a set S given by a constraint $q(y) \leq w$, SCIP can also generate intersection cuts for quadratic equations implied by the condition $X = xx^\top$ (see Sect. 2.3.3). Since X needs to have rank 1, any 2×2 minor of X needs to have determinant zero. Therefore, 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 $X_{i_1 j_1} X_{i_2 j_2} = X_{i_1 j_2} X_{i_2 j_1}$ needs to hold. If all variables in this condition exist in ($\text{MINLP}_{\text{ext}}$), then the procedure to generate intersection cuts is applied to the set defined by this condition, if it is violated.

Since intersection cuts can be rather dense, it is not clear yet how to decide when it will be beneficial to generate such cuts. Their separation is therefore currently disabled by default. For more details, see [17].

2.3.6 Edge-concave cuts

Another method to obtain a linear outer-approximation for a quadratic constraint is by utilizing an edge-concave decomposition of the quadratic function. This has shown to be particularly useful for randomly generated quadratic instances [45, 46]. A function is edge-concave over the variables' domain (e.g., $[\underline{x}, \overline{x}]$) if it is componentwise concave.

Given a quadratic function, the separator for edge-concave cuts solves an auxiliary MIP to partition the square and bilinear terms into a sum of edge-concave functions and a remaining function. Since the convex envelope of edge-concave functions is *vertex-polyhedral* [47], that is, it is a polyhedral function with vertices corresponding to the vertices of the box of variable bounds, facets on the convex envelope of each edge-concave function can be computed by solving an auxiliary linear program (see also Sect. 2.4.1). For the remaining terms, linear underestimators such as (1) are summed up.

Since the current implementation of edge-concave cuts in SCIP has not shown to be particularly useful for general MINLP, it is disabled for now.

2.3.7 Second-order cones

An important connection between MINLP and conic programming is the detection of constraints that can be represented as a second-order cone (SOC) constraint, since the latter defines a convex set, while the original constraint may use a nonconvex constraint function. Thus, SOC detection is the aim of a specialized nonlinear handler in SCIP. In the detection phase, a constraint $h_i(x) \leq w_i$ (the case $\geq$ is handled similarly) of ($\text{MINLP}_{\text{ext}}$) is passed to the nonlinear handler. For this constraint, it is checked whether it defines a bound on an Euclidian norm ($\sqrt{\sum_{j=1}^{k}(a_j y_j^2 + b_j y_j)} + c \leq w_i$ for some coefficients $a_j, b_j, c \in \mathbb{R}$, $a_j > 0$, where y_j is either an original variable or some subexpression of $h_i(\cdot)$), or is a quadratic constraint that is SOC-representable [48]. Since the introduction of slack variables w_i, $i \leq m$, may prevent such a detection, the equivalent constraint $h_i(x) \leq \bar{w}_i$ is considered instead.

Once a SOC constraint has been detected, a solution that violates this constraint can be separated. However, if the detected cone is of high dimension, then many cuts may be required

to provide a tight linear relaxation. Thus, a disaggregation into three-dimensional cones as suggested by Vielma [49] is used.

2.4 Convexity

2.4.1 Convex and concave constraints

For the linear underestimation of functions like $x\exp(x)$ or $x^2+2xy+y^2$, the construction of an extended formulation $(xw, \exp(x) = w; w_1 + 2w_2 + w_3, w_1 = x^2, w_2 = xy, w_3 = y^2)$ is not advisable. Instead, hyperplanes that support the epigraph of a convex function can be used if convexity is recognized. In SCIP, specialized nonlinear handlers are available to detect for a function $h_i(x)$ of ($\text{MINLP}_{\text{ext}}$) the subexpressions that need to be replaced by auxiliary variables $w_{i+1}, \ldots, w_{\hat{m}}$ such that the remaining expression $h_i(x, w_{i+1}, \ldots, w_{\hat{m}})$ is convex or concave. The detection utilizes the often applied rules for convexity/concavity of function compositions (e.g., f convex and monotone decreasing, g concave $\Rightarrow f \circ g$ convex), but applies them in reverse order. That is, instead of deciding whether a function is convex/concave based on information on the convexity/concavity and monotonicity of its arguments, the algorithm formulates conditions on the convexity/concavity of the function arguments given a convexity/concavity requirement on the function itself. When a condition on an argument cannot be fulfilled, it is replaced by an auxiliary variable.

Next to "myopic" rules for convexity/concavity that are implemented by the expression handlers, also rules for product compositions, signomials, and quadratic forms are available. Further, it has been shown that for a composition of convex functions $f \circ g$, it can be beneficial for the linear relaxation to consider the extended formulation $f(w), w \geq g(x)$, instead of the composition $f(g(x))$ [50]. This is enforced by a small variation of the detection algorithm.

When a convex constraint $h_i(x, w_{i+1}, \ldots, w_{\hat{m}}) \leq w_i$ of ($\text{MINLP}_{\text{ext}}$) is violated at a point $(\hat{x}, \hat{w})$, a tangent on the graph of h_i at $(\hat{x}, \hat{w})$ provides a separating hyperplane. If, however, h_i is univariate, that is, $h_i(x, w_{i+1}, \ldots, w_{\hat{m}}) = f(y)$ for some variable y, and y is integral, then taking the hyperplane through the points $(\lfloor\hat{y}\rfloor, f(\lfloor\hat{y}\rfloor))$ and $(\lfloor\hat{y}\rfloor + 1, f(\lfloor\hat{y}\rfloor + 1))$ gives a tighter underestimator.

For a concave function $h_i(x, w_{i+1}, \ldots, w_{\hat{m}})$, any hyperplane $\alpha x + \beta w + \gamma$ that underestimates $h_i(x, w_{i+1}, \ldots, w_{\hat{m}})$ in all vertices of the box $[\underline{x}, \overline{x}] \times [\underline{w}_{i+1}, \overline{w}_{i+1}] \times \cdots \times [\underline{w}_{\hat{m}}, \overline{w}_{\hat{m}}]$ is a valid linear underestimator, since h_i is vertex-polyhedral with respect to the box. Maximizing $\alpha\hat{x} + \beta\hat{w} + \gamma$ such that $\alpha x + \beta w + \gamma$ does not exceed $h_i(x, w_{i+1}, \ldots, w_{\hat{m}})$ for all vertices gives an underestimator that is as tight as possible at a given reference point $(\hat{x}, \hat{w})$. Since the size of this cut generating LP is exponential in k, underestimators for concave functions in more than 14 variables are currently not computed.

2.4.2 Tighter gradient cuts

The separating hyperplanes generated for convex functions of ($\text{MINLP}_{\text{ext}}$) as discussed in the previous section are, in general, not supporting for the feasible region of ($\text{MINLP}_{\text{ext}}$), because the point where the functions are linearized is not at the boundary of the feasible region. Therefore, often several rounds of cut generation and LP solving are required until the relaxation solution satisfies the convex constraints. Solvers for convex MINLP have handled this problem in various ways [7, 51], but the basic idea is to build gradient cuts at a suitable boundary point of the feasible region.

In SCIP, three procedures for building tighter and/or deeper gradient cuts for convex relaxations are included. The first two methods compute a point on the boundary of the set defined by all convex constraints of (MINLP) that is close to the point to be separated [29]. The first method solves an additional nonlinear program to project the point to be separated onto the convex set. Since solving an NLP for every point to be separated can be quite expensive, the second method, going back to an idea by Veinott [52], does a binary search between an interior point of the convex set and the point to be separated. The interior point is computed once in the beginning of the search by solving an auxiliary NLP. The third method does not aim to separate a given point, but utilizes the feasible points that are found by primal heuristics of SCIP. When a new solution is found, gradient cuts are generated at this solution for convex constraints of ($\text{MINLP}_{\text{ext}}$) and added to the cutpool. If such a cut is later found to separate the relaxation solution, it is added to the LP.

All methods are currently disabled as they are not yet efficient in general.

2.5 Quotients

Note that SCIP does not include a dedicated expression handler for quotients, since they can equivalently be written using a product and a power expression. Therefore, 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. However, tighter linear estimates are often possible. Therefore, a specialized nonlinear handler checks whether a given function $h_i(x)$ can be cast as $f(y) = \frac{ay_1+b}{cy_2+d} + e$ with $a, b, c, d, e \in \mathbb{R}$, $a, c \neq 0$, and y_1 and y_2 being either original variables or subexpressions of $h_i(x)$. By distinguishing a number of cases, linear estimators are computed, e.g., by exploring vertex-polyhedrality or by using a formula from [53]. In the univariate case ($y_1 = y_2$), f is either convex or concave if $-d/c \notin [\underline{y}_2, \overline{y}_2]$ and a specialized domain propagation method is used to avoid the dependency problem of interval arithmetic.

2.6 Perspective strengthening

Perspective reformulations have shown to efficiently tighten relaxations of convex mixed-integer nonlinear programs with on/off-structures, which are often modeled via big-M constraints or semi-continuous variables [54]. A variable x_j is semi-continuous with respect to the binary indicator variable $x_{j'}$ if it is fixed to a value x_j^0 when $x_{j'} = 0$ and restricted to a domain $[\underline{x}_j^1, \overline{x}_j^1]$ when $x_{j'} = 1$.

In SCIP, a strengthening of under- and overestimators for functions that depend on semi-continuous variables is available. Consider a constraint $h_i(x, w_{i+1}, \ldots, w_{\hat{m}}) \lesseqgtr w_i$ of ($\text{MINLP}_{\text{ext}}$). A strengthening of under- or overestimators for $h_i(x, w_{i+1}, \ldots, w_{\hat{m}})$ is attempted if the variables that h_i depend on are semi-continuous with respect to the same indicator variable $x_{j'}$.

To determine whether a variable is semi-continuous, suitable bounds that are implied by fixing the same binary variable are searched for. The implied bounds can be obtained either from linear constraints directly or by probing, and are stored by SCIP in a globally available data structure. In addition, an auxiliary variable w_i is found to be semi-continuous if function $h_i(x, w_{i+1}, \ldots, w_{\hat{m}})$ depends only on semi-continuous variables with the same indicator variable.

Assume that a linear underestimator $\ell(x, w_{i+1}, \ldots, w_{\hat{m}})$ has been computed for $h_i(x, w_{i+1}, \ldots, w_{\hat{m}})$. The perspective strengthening extends the underestimator such that it is tight for $x_{j'} = 0$:

$$\ell(x, w_{i+1}, \ldots, w_{\hat{m}}) + \left(h_i(x^0, w^0_{i+1}, \ldots, w^0_{\hat{m}}) - \ell(x^0, w^0_{i+1}, \ldots, w^0_{\hat{m}})\right)(1 - x_{j'}).$$

This extension ensures that the estimator is equal to $h_i(x, w_{i+1}, \ldots, w_{\hat{m}})$ for $x_{j'} = 0$, $(x, w) = (x^0, w^0)$, and equal to $\ell(x, w_{i+1}, \ldots, w_{\hat{m}})$ for $x_{j'} = 1$. If h_i is convex, cuts obtained this way are equivalent to the classic perspective cuts [54]. The method is also applicable when there is a linear part of h_i that depends on variables that are not semi-continuous and that do not appear in the nonlinear part. For more details on the implementation in SCIP, see [19].

2.7 Optimization-based bound tightening

Optimization-Based Bound Tightening (OBBT) is a domain propagation technique which minimizes and maximizes each variable over the feasible set of the problem or a relaxation thereof [55]. Whereas FBBT (see Sect. 2.1.2) propagates the nonlinearities individually, OBBT considers (a relaxation of) all constraints together, and may hence compute tighter bounds, with higher effort.

In SCIP, OBBT solves for each variable x_k that could be subject to spatial branching two LPs that minimize and maximize the variable with respect to the constraints of the LP relaxation and the objective cutoff constraint $c^\top x \leq U$. The optimal values of these LPs may then be used to tighten the bounds of x_k.

By default, OBBT is applied at the root node to tighten bounds globally. It restricts the computational effort by limiting the number of iterations spent for solving the auxiliary LPs and interrupting for cheaper domain propagation techniques between LP solves. Further, the dual solutions of the auxiliary LPs are used to derive linear inequalities that serve as computationally cheap approximation of OBBT during the branch-and-bound search. These inequalities are propagated whenever bounds of variables in the inequality become tighter or a new primal solution is found. For further details, see [56].

In addition to OBBT with respect to the LP relaxation, also a variant is available that optimizes variables with respect to the potentially tighter convex NLP relaxation that is given by all linear and convex nonlinear constraints of (MINLP) [29]. Because of the potentially high computational cost of solving many NLPs, this variant of OBBT is deactivated by default.

2.8 Primal heuristics

The purpose of primal heuristics is to find high quality feasible solutions early in the search. When given an MINLP, up to 40 primal heuristics are active in SCIP by default. Many of them aim to find an integer-feasible solution to the LP relaxation. In the following, primal heuristics that are only active in the presence of nonlinear constraints are discussed.

2.8.1 subNLP

A primal heuristic like `subNLP` is implemented in virtually any MINLP solver. Given a point $\tilde{x}$ that satisfies the integrality requirements, the heuristic fixes all integer variables in

(MINLP) to the values given by $\tilde{x}$. It then calls the SCIP presolver on this subproblem for possible simplifications. Finally, it triggers a solution of the remaining NLP, using $\tilde{x}$ as the starting point. If the NLP solver, such as Ipopt, finds a solution that is feasible (and often also locally optimal) for the NLP relaxation, then a feasible point for (MINLP) has been found.

The starting point $\tilde{x}$ can be the current solution of the LP relaxation if integer-feasible, a point found by a primal heuristic that searches for integer-feasible solutions of the LP relaxation, or a point that is passed on by other primal heuristics for MINLP, such as those mentioned in the next sections.

2.8.2 Multistart

If (MINLP) is nonconvex after fixing all integer variables, then several local optima may exist for the NLPs solved by heuristic `subNLP`. Depending on the starting point, the NLP solver may find different local optimum. Therefore, the multistart heuristic aims to compute several starting points for `subNLP`.

The algorithm, originally developed in [57], aims to approximate the boundary of the feasible set of the NLP relaxation by sampling points from $[\underline{x}, \overline{x}]$ and pushing them towards the feasible set by the use of an inexpensive gradient descent method. Afterwards, points that are relatively close to each other are grouped into clusters. Ideally, each cluster approximates the boundary of some connected component of the feasible set. For each cluster, a linear combination of the points is passed as a starting point to `subNLP`. For integer variables, the value in the starting point is rounded to an integral value. However, since this most likely leads to infeasible NLPs, the multistart heuristic currently runs for continuous problems only by default. For more details, see [29].

2.8.3 NLP diving

As an alternative to finding a good fixing for all integer variables of (MINLP), the NLP diving heuristic starts by solving the NLP relaxation at the current branch-and-bound node with an NLP solver, using the solution of the LP relaxation as starting point. It then iteratively fixes integer variables with fractional value and resolves both the LP and NLP relaxations, thereby simulating a depth-first-search in a branch-and-bound tree. By default, variables for which the sum of the distances from the solutions of the LP and NLP relaxations to a common integer value is minimal are rounded to the nearest integer value. Further, binary and nonlinear variables are preferred. If the resulting NLP is found to be (locally) infeasible, one-level backtracking is applied, that is, the last fixing is undone, and the opposite fixing is tried.

2.8.4 MPEC

While the NLP diving heuristic either completely omits or enforces integrality restrictions in the NLP relaxation, the MPEC heuristic adds a relaxation of the integrality restriction to the NLP and tightens this relaxation iteratively. The heuristic is only applicable to mixed-binary nonlinear programs at the moment.

The basic idea of the heuristic, originally developed in [58], is to reformulate (MINLP) as a mathematical program with equilibrium constraints (MPEC) and to solve this MPEC to local optimality. The MPEC is obtained by rewriting the condition $x_i \in \{0, 1\}$, $i \in \mathcal{I}$, as complementarity constraint $x_i \perp 1 - x_i$. This reformulation is again reformulated to an NLP

by writing it as $x_i\ (1-x_i) = 0$. However, these reformulated complementarity constraints will not, in general, satisfy constraint qualifications. Therefore, in order to increase the chances of solving the NLP reformulation, the heuristic solves regularized versions of the NLP by relaxing $x_i(1 - x_i) = 0$ to $x_i(1 - x_i) \leq \theta$, for different, ever smaller $\theta > 0$. If the NLP solution is close to satisfying $x_{\mathcal{I}} \in \{0, 1\}^{|\mathcal{I}|}$, it is passed as starting point to the `subNLP` heuristic. If an NLP is (locally) infeasible, the heuristic does two more attempts where the values for binary variables that are already close to 0 or 1 are flipped to 1 or 0, respectively. For more details, see [32].

2.8.5 Undercover

While the previous heuristics focused on enforcing the integrality condition on an NLP, heuristic `undercover` [59] starts from a completely different angle. The heuristic is based on the observation that it sometimes suffices to fix only a comparatively small number of variables of (MINLP) to yield a mixed-integer linear subproblem. For example, for a bilinear term, only one of the variables needs to be fixed. A set covering problem is solved to minimize the number of variables to fix. The values for the fixed variables are taken from solutions of the LP or NLP relaxation or a known feasible solution of the MINLP.

The resulting sub-MIP is less complex to solve, and does not need to be solved to proven optimality. The solutions of the sub-MIP are immediately feasible for (MINLP). However, the best one is also passed as starting point to heuristic `subnlp` to try for further improvement. For more details, see [59].

3 Benchmark

This section aims to present a fair comparison of SCIP with several other state-of-the-art solvers for general MINLP. Doing so is not trivial at all. First, a set of instances needs to be selected that is suitable as a benchmark set. Second, solver parameters have to be set such that all solvers solve the same instances with the same working limits and the same requirements on feasibility and optimality—this goal could not be reached completely. Third, the solver's results have to be checked for correctness, or, when this is not possible, plausibility.

GAMS was used for the experiments, as it provides various facilities to help on solver comparisons and comes with current versions of SCIP and the commercial solvers BARON [60], Lindo API [61], and Octeract included.

All computations were run on a Linux cluster with Intel Xeon E5-2670 v2 CPUs (20 cores). The GAMS version is 41.2.0, which includes SCIP 8.0.2, BARON 22.9.30, Lindo API 14.0.5099.162, and Octeract 4.5.1. A GAMS license with all solvers enabled was used, so that SCIP uses CPLEX 22.1.0.0 as LP solver and Ipopt with HSL MA27 as NLP solver, BARON can choose between all LP/MIP/NLP solvers that it interfaces with, and Octeract uses CPLEX 22.1.0.0 as LP/MIP/QP/QCP solver.

3.1 Test set

To construct a test set suitable for benchmarking, the MINLPLib [62] collection of 1595 MINLPs was used as source. First, instances that could not be handled by some solver were excluded. All solvers were then run on the remaining 1505 instances using the parameter settings described below. The results of these runs were then used to select 200 instances

that could be solved by at least one solver, that were not trivial for all solvers, had a varying degree of integrality and nonlinearity, and such that having many instances with a similar name is avoided. The latter was done to avoid overrepresentation of problems for which many instances were added to MINLPLib.

Since small changes to an instance can lead to large variations in the solver's performance, the benchmark's reliability is improved by considering for each instance four additional variants where the order of variables and equations has been permuted. Thus, a test set of 1000 instances is obtained.

The following approach was used to select the benchmark set of 200 instances before permutation: Let I be the initial set of 1505 instances from MINLPLib, d_i be the fraction of integer variables in instance $i \in I$, and e_i be the fraction of nonzeros in the Jacobian and objective function gradient that correspond to nonlinear terms. Next, assign to each instance an identifier $f_i \in F$ such that instances that seem to come from the same model are assigned the same identifier. This goal is approximated by mapping i to the name of the instance until the first digit, underscore, or dash, except for the block layout design instances `fo*`, `m*`, `no*`, `o*`, which were all assigned to the same identifier. $|F| = 230$ different identifiers were found this way.

Further, let $\bar{t}_i$ be the largest time in seconds that any solver who did not produce wrong results on instance i spend on instance i. Finally, let S be the number of instances that could be solved by at least one solver.

To ensure that instances with a varying amount of integer variables and nonlinearity are included, the interval $[0, 1]$ was split once at breakpoints $0.05, 0.25, 0.5, 0.9$ and once at $0.1, 0.25, 0.5$. Let D and E be the resulting partitions of $[0, 1]$. For every interval from D and E, the aim is to have roughly the same number of instances with d_i and e_i in the respective intervals. For the choice of breakpoints that define D and E, the distribution of d_i and e_i, $i \in I$, have been taken into account. For example, MINLPLib contains many purely continuous and purely discrete instances, but not many instances that are mostly linear or completely nonlinear.

To avoid including too many instances originating from the same model, including more than two instances for each identifier in F is discouraged. Further, instances that seem trivial, i.e., which are solved by all solvers in no more than five seconds, or could not be solved by any solver are excluded. Introducing penalty terms, the following optimization problem for instance selection is obtained:

$$
\begin{aligned}
\min \quad & \sum_{d \in D} \lambda_d^2 + \sum_{e \in E} \lambda_e^2 + 10 \sum_{f \in F} \lambda_f^2 \\
\text{such that} \quad & \sum_{i \in I : d_i \in d} z_i = \left\lfloor \frac{N}{|D|} \right\rceil + \lambda_d && \forall d \in D, \\
& \sum_{i \in I : e_i \in e} z_i = \left\lfloor \frac{N}{|E|} \right\rceil + \lambda_e && \forall e \in E, \\
& \sum_{i \in I : f_i = f} z_i \leq 2 + \lambda_f && \forall f \in F, \\
& z_i = 0 && \forall i \in I : \bar{t}_i \leq 5, \\
& z_i = 0 && \forall i \in I : i \notin S, \\
& z \in \{0, 1\}^{|I|}, \lambda \in \mathbb{Z}^{|D|+|E|+|F|}
\end{aligned}
$$

Table 1 Number of instances selected with "discreteness" d_i and "nonlinearity" e_i in intervals from D and E

$E \downarrow \mid D \rightarrow$	[0,0.05)	[0.05,0.25)	[0.25,0.5)	[0.5,0.9)	[0.9,1]	[0,1]
[0, 0.1)	3	7	19	15	6	50
[0.1, 0.25)	8	22	9	7	4	50
[0.25, 0.5)	8	8	6	10	18	50
[0.5, 1]	25	2	5	7	11	50
[0, 1]	44	39	39	39	39	200

This problem was solved for N varying between 180 and 220. For $N = 208$, this yield a selection of 200 instances with an acceptable penalty value of 106. Table 1 shows the number of instances for each element of $D \times E$. For five identifiers from F, three instead of two instances were selected, i.e., $\lambda_f = 1$ for five $f \in F$. Section 1 in the supplement gives the list of selected instances.

3.2 Parameter settings

3.2.1 Missing variable bounds

To compute a lower bound on the optimal value of a minimization problem, all solvers considered here construct a convex relaxation of the given problem. For nonconvex constraints, this often relies on the computation of valid convex underestimators or concave overestimators. As these typically depend on variables' bounds (recall (1)), an instance with missing or very large bounds on variables in nonconvex terms can be very hard or impossible to solve.

Even when the user forgot to specify some variable bounds, the solver may still be able to derive bounds via domain propagation. Further, once a feasible solution $\hat{x}$ has been found, additional bounds may be derived from the inequality $c^\top x \leq c^\top \hat{x}$. However, as there are always cases where bounds are still missing after presolve, solvers invented different ways to deal with this obstacle.

If SCIP cannot under- or overestimate because of missing variable bounds, it continues by branching on an unbounded variable. This way, there will eventually be a node in the branch-and-bound tree where all variables are bounded. Nodes that still contain unbounded variable domains may be pruned due to a derived lower bound on the objective function exceeding the incumbents objective function value. But it may also be the case that pruning will not be possible and SCIP does not terminate. However, variable bounds after branching cannot grow indefinitely in SCIP, but are limited by $\pm 10^{20}$ by default. That is, SCIP does not search for solutions with variable values beyond this value.

The other solvers considered here add variable bounds based on a heuristic decision. If BARON is still missing bounds on variables in nonconvex terms after presolve, it sets the bound to a value that depends on the type of nonlinearity involved. Typically, this value is around $\pm 10^{10}$. BARON also prints a warning and no longer claim to have solved a problem to global optimality, i.e., it does not return a lower bound. Lindo API adjusts the bounds for all variables that are involved in convexification to be within $[-10^{10}, 10^{10}]$. At termination, it returns the lower bound for the restricted problem. Octeract proceeds similarly and introduces a bound of $\pm 10^7$ for every missing bound and returns the lower bound for the restricted problem at termination.

Evidently, passing an instance with unbounded variables to several solvers with default settings may mean that each solver solves a different subproblem of the actual problem and often also reports a lower bound that corresponds to the solved subproblem only. Fortunately, parameters are available to adjust the treatment of unbounded variables. A first impulse could be to tell all solvers to set missing bounds to infinity, but this is not possible as each solver treats values beyond a different finite value as "infinity" (BARON: 10^{50}, Octeract: 10^{308}, SCIP: 10^{20}). Changing this value is either not possible or not advisable.

We therefore decided to aim for $\pm 10^{12}$ as replacement for a missing variable bound. For BARON and SCIP, the GAMS interface can replace any missing bound by $\pm 10^{12}$ before the instance is passed to the solver. BARON will hence also return a lower bound for this restricted problem. For Lindo API, a solver parameter can be changed so that bounds for all variables subject to convexification are bounded by $\pm 10^{12}$ (instead of $\pm 10^{10}$). Finally, also for Octeract, all missing bounds are set to $\pm 10^{12}$ (instead of $\pm 10^{7}$) by changing of a solver parameter. Note, that this still does not ensure that all solvers solve the same instance, since Lindo API may still change initial finite bounds beyond 10^{12} and may not bound variables that are not involved in convexification.

Next to missing bounds on problem variables, also singularities in functions (e.g., $1/x$, $\log(x)$) can make finite estimators unavailable. Unfortunately, there are no parameters available to ensure a uniform treatment of this case in all solvers. SCIP ensures that the variable in x^p, $p < 0$, or $\log(x)$ is bounded away from zero by 10^{-9}, and terminates with a lower bound for this modified problem. BARON applies the same method as the one for missing variable bounds to choose a suitable bound on x. No lower bound is returned at termination then. The methods in Lindo API and Octeract are not known to us.

3.2.2 Solution quality

To ensure that all solvers return solutions of the same quality, constraints of (MINLP) are required to be satisfied with an absolute tolerance of 10^{-6}. This applies to linear and nonlinear equations, variable bounds, and integrality.

In addition, a tolerance on the proof of optimality is set. For this purpose, typically, solvers are allowed to stop when the absolute or relative gap between lower and upper bounds on the optimal value are sufficiently small. Since the test set is diverse and has optimal values of varying magnitude, setting only a relative gap limit and no absolute gap limit would be preferable. Unfortunately, Octeract does not permit different values for these limits. As a compromise, BARON, Lindo API, and SCIP are run with 10^{-4} as relative gap limit and 10^{-6} as absolute gap limit, while for Octeract, 10^{-6} is used for both the absolute and relative gap limit. Section 2.2 in the supplement shows that this tighter optimality tolerance has essentially no effect on the performance of Octeract.

3.2.3 Working limits

As working limits, a time limit of two hours is used and the jobs on the cluster are restricted to 50 GB of RAM. Further, parallelization functionality has been disabled. For a comparison with parallelization enabled, see [14].

3.2.4 Summary

To summarize, the following parameters are used:

Table 2 Aggregated performance data for all solvers on test set of 1000 instances

	Solved	Timeout	Fail	Best obj	Time
BARON	790	183	27	928	75.4
Lindo API	538	323	139	729	489.1
Octeract	671	279	50	848	184.1
SCIP	776	183	41	922	85.2
Virt. worst	368	405	227	589	1505.2
Virt. best	967	33	0	987	19.7

GAMS (applied to all solvers): `optcr=1e-4`, `optca=1e-6`, `reslim=7200`, `workspace=50000`, `threads=1`
BARON: `InfBnd=1e12`, `AbsConFeasTol=1e-6`, `AbsIntFeasTol=1e-6`
Lindo API: `GOP_BNDLIM=1e12`, `SOLVER_FEASTOL=1e-6`
Octeract: `INFINITY=1e12`, `INTEGRALITY_VIOLATION_TOLERANCE=1e-6`
SCIP: `gams/infbound=1e12`, `constraints/nonlinear/linearizeheursol=o` (this undoes a change in the algorithmic settings of SCIP that is part of the GAMS/SCIP interface)

3.3 Correctness checks

A run of a solver on an instance is marked as *failed* if the solver terminated abnormally, the solution is not feasible with respect to the feasibility tolerance, or the lower or upper bound contradicts with the bounds on the optimal value that are specified on the MINLPLib page.

A run that has not failed is marked as *solved* if the relative or absolute gap limits are satisfied. If a solver stopped without closing the gap before the time limit, then the solver time is changed to the time limit. The only exception here is BARON, which stops on two instances before the time limit without reporting a lower bound due to singularities in functions (see Sect. 3.2.1). To be consistent with the treatment of other solvers, these two instances were accounted as solved by BARON with the original solver time.

3.4 Results

Table 2 shows for each solver the number of instances that could be solved, how often the time limit was reached, and the number of runs that were marked as failed. In addition, the number of instances for which a solution with objective value not more than 1% worse than the best solution found by any considered solver is shown. Finally, the shifted geometric mean of the running time of the solver is provided. The shift has been set to 1 s. Here, instances that failed are accounted with the time limit. In addition, results for the *virtual best* and *virtual worst* solver are reported, which are obtained by picking for each instance the fastest or slowest solver (best or worst objective function value for "best obj." column), respectively. The performance profile in Fig. 1 shows the number of instances a solver solved with a time that is at most a factor of the fastest solvers time. Section 2.1 in the supplement provides detailed results.

The results show a small lead of BARON before SCIP with respect to the number of instances solved, number of instances finding a best solution, and average time. Since the number of timeouts is almost equal, one could argue that it is the higher stability of BARON

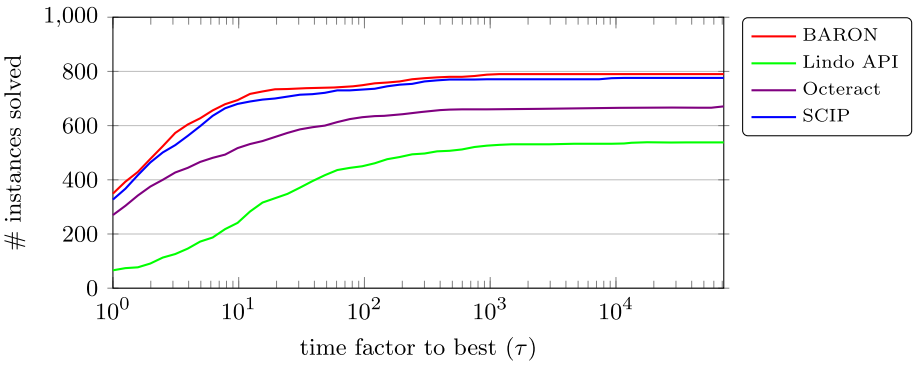

Fig. 1 Performance profile comparing all solvers

that moves it onto the first place here. In fact, the 41 fails of SCIP are due to returning a wrong optimal value 16 times, returning an infeasible solution 23 times, and aborts due to numerical troubles for two instances. For BARON, fails are due to returning a wrong optimal value 26 times and an infeasible solution only once. While SCIP 8 has made a large step forward in ensuring that nonlinear constraints are satisfied in the non-presolved problem, violations in linear constraints or variable bounds still occur for a few instances. These are typically due to variables being aggregated during presolve.

Even though Octeract and Lindo API solved considerably fewer instances than BARON and SCIP, which also results in an increased mean time, it is noteworthy that each of the two is also the fastest solver on 270 and 66 instances, respectively. Octeract also produced correct results for 95% of the test set, while for Lindo API a relatively high number of wrong optimal values, infeasible solutions, or aborts is observed.

The large differences between the real and virtual solvers show that none of the solvers dominates all others or is dominated.

4 Conclusion

The development of the MINLP solver in SCIP has come a long way. In a recent version-to-version comparison [13, slides 49–51], a steady improvement in the performance of SCIP on MINLP over the last ten years has been measured, resulting in SCIP 8 solving twice as many instances as SCIP 3 and a speed-up of factor three. Partially, this improvement has been achieved by improving and adding features particular for MINLP. However, due to the generality of SCIP as a CIP solver, also many developments that targeted MIP solving were immediately available for MINLP solving.

With version 8, the MINLP solving capabilities of SCIP have been largely reworked and extended, which resulted in a considerable improvement in both robustness and performance [13, 16]. As a result, SCIP's performance is currently on par with the state-of-the-art commercial solver BARON.

In contrast to the commercial solvers considered here, SCIP offers a variety of possibilities for a user, developer, or researcher to interact with the solving process. In particular, the newly added "nonlinear handler" plugin type sets SCIP apart from most other MINLP solvers, as it allows focusing on experimenting with new algorithms to handle certain structures in nonlinear functions without modifying the solver's code.

The rather large number of features that are disabled by default shows that tuning and improving the existing code base has become increasingly necessary. Of course, also new features will be added in the future, e.g., improved separation for signomial functions [63], alternative relaxations for polynomial functions [64], or monoidal strengthening of intersection cuts for quadratics [65].

Supplementary information

A supplement with all data generated or analyzed for the computational experiments during this study is available online.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s10898-023-01345-1.

Acknowledgements We are in all SCIP developers' debt—the extensions to support nonlinear constraints and solve MINLPs would not have been possible without the framework's existence and the powerful MIP solver that we could build upon. While the authors of this paper are the main developers of the new MINLP features in SCIP 8, many have contributed to the MINLP capabilities in previous releases of SCIP, namely Martin Ballerstein, Timo Berthold, Tobias Fischer, Thorsten Gellermann, Ambros Gleixner, Renke Kuhlmann, Dennis Michaels, Marc Pfetsch, and Stefan Weltge. Last but not least, we are very grateful to Franziska Schlösser for the setup and maintenance of benchmarking and testing facilities for the infamous "consexpr" development branch of SCIP.

Funding Open Access funding enabled and organized by Projekt DEAL. The work for this article has been conducted within the Research Campus Modal funded by the German Federal Ministry of Education and Research (BMBF Grant Numbers 05M14ZAM, 05M20ZBM). Additional funding has been received from the German Federal Ministry for Economic Affairs and Energy within the project EnBA-M (ID: 03ET1549D).

Declarations

Conflict of interest The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Floudas, C.A.: Nonlinear and Mixed Integer Optimization: Fundamentals and Applications. Oxford University Press, New York (1995)
2. Grossmann, I.E., Kravanja, Z.: Mixed-integer nonlinear programming: a survey of algorithms and applications. In: Conn, A.R., Biegler, L.T., Coleman, T.F., Santosa, F.N. (eds.) Large-Scale Optimization with Applications, Part II: Optimal Design and Control, pp. 73–100. Springer, New York (1997). https://doi.org/10.1007/978-1-4612-1960-6_5
3. Pintér, J.D. (ed.): Global Optimization: Scientific and Engineering Case Studies Nonconvex Optimization and Its Applications, vol. 85. Springer, New York (2006). https://doi.org/10.1007/0-387-30927-6

4. Trespalacios, F., Grossmann, I.: Review of mixed-integer nonlinear and generalized disjunctive programming methods. Chemie Ingenieur Technik **86**(7), 991–1012 (2014). https://doi.org/10.1002/cite.201400037
5. Bussieck, M.R., Vigerske, S.: MINLP solver software. In: Cochran, J.J., Cox, L.A., Jr., Keskinocak, P., Kharoufeh, J.P., Smith, J.C. (eds.) Wiley Encyclopedia of Operations Research and Management Science. Wiley, Hoboken (2010). https://doi.org/10.1002/9780470400531.eorms0527
6. Kocis, G.R., Grossmann, I.E.: Computational experience with DICOPT: solving MINLP problems in process systems engineering. Comput. Chem. Eng. **13**(3), 307–315 (1989). https://doi.org/10.1016/0098-1354(89)85008-2
7. Duran, M.A., Grossmann, I.E.: An outer-approximation algorithm for a class of mixed-integer nonlinear programs. Math. Programm. **36**(3), 307–339 (1986). https://doi.org/10.1007/BF02592064
8. Adjiman, C.S., Floudas, C.A.: Rigorous convex underestimators for general twice-differentiable problems. J. Glob. Optim. **9**(1), 23–40 (1996). https://doi.org/10.1007/BF00121749
9. Sahinidis, N.V.: BARON: a general purpose global optimization software package. J. Glob. Optim. **8**(2), 201–205 (1996). https://doi.org/10.1007/BF00138693
10. Smith, E.M.B., Pantelides, C.C.: A symbolic reformulation/spatial branch-and-bound algorithm for the global optimisation of nonconvex MINLPs. Comput. Chem. Eng. **23**(4–5), 457–478 (1999). https://doi.org/10.1016/s0098-1354(98)00286-5
11. Vigerske, S., Gleixner, A.: SCIP: global optimization of mixed-integer nonlinear programs in a branch-and-cut framework. Optim. Methods Softw. **33**(3), 563–593 (2017). https://doi.org/10.1080/10556788.2017.1335312
12. Achterberg, T.: Constraint Integer Programming. PhD thesis, Technische Universität Berlin (2007)
13. Pfetsch, M.: SCIP: past, present, future. Presentation at workshop *Let's SCIP it!* (2022). https://scipopt.org/20years/slides/pfetsch.pdf
14. Bestuzheva, K., Chmiela, A., Müller, B., Serrano, F., Vigerske, S., Wegscheider, F.: Global optimization of mixed-integer nonlinear programs with SCIP 8.0. Technical report (2022). https://optimization-online.org/?p=21314
15. Belotti, P., Kirches, C., Leyffer, S., Linderoth, J., Luedtke, J., Mahajan, A.: Mixed-integer nonlinear optimization. Acta Numer. **22**, 1–131 (2013). https://doi.org/10.1017/S0962492913000032
16. Bestuzheva, K., Besançon, M., Chen, W.-K., Chmiela, A., Donkiewicz, T., Doornmalen, J., Eifler, L., Gaul, O., Gamrath, G., Gleixner, A., Gottwald, L., Graczyk, C., Halbig, K., Hoen, A., Hojny, C., Hulst, R., Koch, T., Lübbecke, M., Maher, S.J., Matter, F., Mühmer, E., Müller, B., Pfetsch, M.E., Rehfeldt, D., Schlein, S., Schlösser, F., Serrano, F., Shinano, Y., Sofranac, B., Turner, M., Vigerske, S., Wegscheider, F., Wellner, P., Weninger, D., Witzig, J.: The SCIP optimization suite 8.0. ZIB report 21–41, Zuse Institute Berlin (2021). nbn:de:0297-zib-85309
17. Chmiela, A., Muñoz, G., Serrano, F.: On the implementation and strengthening of intersection cuts for QCQPs. In: Singh, M., Williamson, D.P. (eds.) Integer Programming and Combinatorial Optimization, pp. 134–147. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-73879-2_10
18. Bestuzheva, K., Gleixner, A., Achterberg, T.: Efficient separation of RLT cuts for implicit and explicit bilinear products. In: Del Pia, A., Kaibel, V. (eds.) Integer Programming and Combinatorial Optimization, pp. 14–28. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-32726-1_2
19. Bestuzheva, K., Gleixner, A., Vigerske, S.: A computational study of perspective cuts. Math. Program. Comput. **15**(4), 703–731 (2023). https://doi.org/10.1007/s12532-023-00246-4
20. Wegscheider, F.: Exploiting symmetry in mixed-integer nonlinear programming. Master's thesis, Zuse Institute Berlin (2019). nbn:de:0297-zib-77055
21. Moore, R.E.: Interval Analysis. Prentice Hall, Englewood Cliffs, NJ (1966)
22. McCormick, G.P.: Computability of global solutions to factorable nonconvex programs: part I—convex underestimating problems. Math. Program. **10**(1), 147–175 (1976). https://doi.org/10.1007/bf01580665
23. Fletcher, R., Leyffer, S.: User manual for filterSQP. Numerical Analysis Report NA/181, Department of Mathematics, University of Dundee, Scotland (1998)
24. Wächter, A., Biegler, L.T.: On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. Math. Program. **106**(1), 25–57 (2006). https://doi.org/10.1007/s10107-004-0559-y
25. Büskens, C., Wassel, D.: The ESA NLP solver WORHP. In: Fasano, G., Pintér, J.D. (eds.) Modeling and Optimization in Space Engineering. Springer Optimization and Its Applications, vol. 73, pp. 85–110. Springer, New York (2013). https://doi.org/10.1007/978-1-4614-4469-5_4
26. Bell, B.: CppAD: a Package for Differentiation of C++ Algorithms. https://github.com/coin-or/CppAD/
27. Hansen, P., Jaumard, B., Ruiz, M., Xiong, J.: Global minimization of indefinite quadratic functions subject to box constraints. Naval Res. Logist. (NRL) **40**(3), 373–392 (1993). https://doi.org/10.1002/1520-6750(199304)40:3<373::AID-NAV3220400307>3.0.CO;2-A

28. Berthold, T., Heinz, S., Pfetsch, M.E.: Nonlinear pseudo-boolean optimization: relaxation or propagation? In: Kullmann, O. (ed.) Theory and Applications of Satisfiability Testing—SAT 2009, pp. 441–446. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_40
29. Maher, S.J., Fischer, T., Gally, T., Gamrath, G., Gleixner, A., Gottwald, R.L., Hendel, G., Koch, T., Lübbecke, M.E., Miltenberger, M., Müller, B., Pfetsch, M.E., Puchert, C., Rehfeldt, D., Schenker, S., Schwarz, R., Serrano, F., Shinano, Y., Weninger, D., Witt, J.T., Witzig, J.: The SCIP optimization suite 4.0. ZIB report 17–12, Zuse Institute Berlin (2017). nbn:de:0297-zib-62170
30. Fischer, T.: Branch-and-cut for complementarity and cardinality constrained linear programs. PhD thesis, Technical University of Darmstadt (2017)
31. Hojny, C., Pfetsch, M.E.: Polytopes associated with symmetry handling. Math. Program. **175**(1), 197–240 (2019). https://doi.org/10.1007/s10107-018-1239-7
32. Gleixner, A., Eifler, L., Gally, T., Gamrath, G., Gemander, P., Gottwald, R.L., Hendel, G., Hojny, C., Koch, T., Miltenberger, M., Müller, B., Pfetsch, M.E., Puchert, C., Rehfeldt, D., Schlösser, F., Serrano, F., Shinano, Y., Viernickel, J.M., Vigerske, S., Weninger, D., Witt, J.T., Witzig, J.: The SCIP optimization suite 5.0. ZIB report 17–61, Zuse Institute Berlin (2017). nbn:de:0297-zib-66297
33. Gamrath, G., Anderson, D., Bestuzheva, K., Chen, W.-K., Eifler, L., Gasse, M., Gemander, P., Gleixner, A., Gottwald, L., Halbig, K., Hendel, G., Hojny, C., Koch, T., Bodic, P.L., Maher, S.J., Matter, F., Miltenberger, M., Mühmer, E., Müller, B., Pfetsch, M.E., Schlösser, F., Serrano, F., Shinano, Y., Tawfik, C., Vigerske, S., Wegscheider, F., Weninger, D., Witzig, J.: The SCIP optimization suite 7.0. ZIB report 20–10, Zuse Institute Berlin (2020). nbn:de:0297-zib-78023
34. Margot, F.: Symmetry in integer linear programming. In: Jünger, M., Liebling, T.M., Naddef, D., Nemhauser, G.L., Pulleyblank, W.R., Reinelt, G., Rinaldi, G., Wolsey, L.A. (eds.) 50 Years of Integer Programming, pp. 647–686. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-540-68279-0_17
35. Liberti, L.: Reformulations in mathematical programming: automatic symmetry detection and exploitation. Math. Program. **131**(1), 273–304 (2012). https://doi.org/10.1007/s10107-010-0351-0
36. Domes, F., Neumaier, A.: Constraint propagation on quadratic constraints. Constraints **15**(3), 404–429 (2010). https://doi.org/10.1007/s10601-009-9076-1
37. Locatelli, M.: Convex envelopes of bivariate functions through the solution of KKT systems. J. Glob. Optim. **72**(2), 277–303 (2018). https://doi.org/10.1007/s10898-018-0626-1
38. Müller, B., Serrano, F., Gleixner, A.: Using two-dimensional projections for stronger separation and propagation of bilinear terms. SIAM J. Optim. **30**(2), 1339–1365 (2020). https://doi.org/10.1137/19m1249825
39. Adams, W.P., Sherali, H.D.: A tight linearization and an algorithm for zero-one quadratic programming problems. Manag. Sci. **32**(10), 1274–1290 (1986). https://doi.org/10.1287/mnsc.32.10.1274
40. Adams, W.P., Sherali, H.D.: Linearization strategies for a class of zero-one mixed integer programming problems. Oper. Res. **38**(2), 217–226 (1990). https://doi.org/10.1287/opre.38.2.217
41. Tuy, H.: Concave programming with linear constraints. Doklady Akademii Nauk **159**(1), 32–35 (1964)
42. Balas, E.: Intersection cuts—a new type of cutting planes for integer programming. Oper. Res. **19**(1), 19–39 (1971). https://doi.org/10.1287/opre.19.1.19
43. Muñoz, G., Serrano, F.: Maximal quadratic-free sets. In: Bienstock, D., Zambelli, G. (eds.) Integer Programming and Combinatorial Optimization, pp. 307–321. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45771-6_24
44. Glover, F.: Polyhedral convexity cuts and negative edge extensions. Z. für Oper. Res. **18**, 181–186 (1974). https://doi.org/10.1007/BF02026599
45. Misener, R., Floudas, C.A.: Global optimization of mixed-integer quadratically-constrained quadratic programs (MIQCQP) through piecewise-linear and edge-concave relaxations. Math. Program. **136**(1), 155–182 (2012). https://doi.org/10.1007/s10107-012-0555-6
46. Misener, R., Smadbeck, J.B., Floudas, C.A.: Dynamically generated cutting planes for mixed-integer quadratically constrained quadratic programs and their incorporation into GloMIQO 2. Optim. Methods Softw. **30**(1), 215–249 (2015). https://doi.org/10.1080/10556788.2014.916287
47. Tardella, F.: On the existence of polyhedral convex envelopes. In: Floudas, C.A., Pardalos, P. (eds.) Frontiers in Global Optimization, pp. 563–573. Springer, Boston (2004). https://doi.org/10.1007/978-1-4613-0251-3_30
48. Mahajan, A., Munson, T.: Exploiting second-order cone structure for global optimization. Technical Report ANL/MCS-P1801-1010, Argonne National Laboratory (2010)
49. Vielma, J.P., Dunning, I., Huchette, J., Lubin, M.: Extended formulations in mixed integer conic quadratic programming. Math. Program. Comput. **9**(3), 369–418 (2016). https://doi.org/10.1007/s12532-016-0113-y

50. Tawarmalani, M., Sahinidis, N.V.: A polyhedral branch-and-cut approach to global optimization. Math. Program. **103**(2), 225–249 (2005). https://doi.org/10.1007/s10107-005-0581-8
51. Kronqvist, J., Lundell, A., Westerlund, T.: The extended supporting hyperplane algorithm for convex mixed-integer nonlinear programming. J. Glob. Optim. **64**(2), 249–272 (2016). https://doi.org/10.1007/s10898-015-0322-3
52. Veinott, A.F.: The supporting hyperplane method for unimodal programming. Oper. Res. **15**(1), 147–152 (1967). https://doi.org/10.1287/opre.15.1.147
53. Zamora, J.M., Grossmann, I.E.: Continuous global optimization of structured process systems models. Comput. Chem. Eng. **22**(12), 1749–1770 (1998). https://doi.org/10.1016/S0098-1354(98)00244-0
54. Frangioni, A., Gentile, C.: Perspective cuts for a class of convex 0–1 mixed integer programs. Math. Program. **106**(2), 225–236 (2006). https://doi.org/10.1007/s10107-005-0594-3
55. Quesada, I., Grossmann, I.E.: Global optimization algorithm for heat exchanger networks. Ind. Eng. Chem. Res. **32**(3), 487–499 (1993). https://doi.org/10.1021/ie00015a012
56. Gleixner, A., Berthold, T., Müller, B., Weltge, S.: Three enhancements for optimization-based bound tightening. J. Glob. Optim. **67**(4), 731–757 (2017). https://doi.org/10.1007/s10898-016-0450-4
57. Smith, L., Chinneck, J., Aitken, V.: Improved constraint consensus methods for seeking feasibility in nonlinear programs. Comput. Optim. Appl. **54**(3), 555–578 (2013). https://doi.org/10.1007/s10589-012-9473-z
58. Schewe, L., Schmidt, M.: Computing feasible points for binary MINLPs with MPECs. Math. Program. Comput. **11**(1), 95–118 (2019). https://doi.org/10.1007/s12532-018-0141-x
59. Berthold, T., Gleixner, A.: Undercover: a primal MINLP heuristic exploring a largest sub-MIP. Math. Program. **144**(1–2), 315–346 (2014). https://doi.org/10.1007/s10107-013-0635-2
60. Khajavirad, A., Sahinidis, N.V.: A hybrid LP/NLP paradigm for global optimization relaxations. Math. Program. Comput. **10**(3), 383–421 (2018). https://doi.org/10.1007/s12532-018-0138-5
61. Lin, Y., Schrage, L.: The global solver in the LINDO API. Optim. Methods Softw. **24**(4–5), 657–668 (2009). https://doi.org/10.1080/10556780902753221
62. A Library of Mixed-Integer and Continuous Nonlinear Programming Instances. https://www.minlplib.org (2022-10-14)
63. Xu, L., D'Ambrosio, C., Liberti, L., Vanier, S.H.: On cutting planes for extended formulation of signomial programming (2022) arXiv:2212.02857
64. Bestuzheva, K., Gleixner, A., Völker, H.: Strengthening SONC relaxations with constraints derived from variable bounds. ZIB-Report 23-03, Zuse Institute Berlin (2023). nbn:de:0297-zib-89510
65. Chmiela, A., Muñoz, G., Serrano, F.: Monoidal strengthening and unique lifting in MIQCPs. In: Del Pia, A., Kaibel, V. (eds.) Integer Programming and Combinatorial Optimization, pp. 87–99. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-32726-1_7