Mathematical Programming Computation (2021) 13:301–338
https://doi.org/10.1007/s12532-020-00196-1

FULL LENGTH PAPER

Minotaur: a mixed-integer nonlinear optimization toolkit

Ashutosh Mahajan[1] · **Sven Leyffer**[2] · **Jeff Linderoth**[3] · **James Luedtke**[3] · **Todd Munson**[2]

Received: 16 October 2017 / Accepted: 19 August 2020 / Published online: 7 November 2020

Abstract

We present a flexible framework for general mixed-integer nonlinear programming (MINLP), called Minotaur, that enables both algorithm exploration and structure exploitation without compromising computational efficiency. This paper documents the concepts and classes in our framework and shows that our implementations of standard MINLP techniques are efficient compared with other state-of-the-art solvers. We then describe structure-exploiting extensions that we implement in our framework and demonstrate their impact on solution times. Without a flexible framework that enables structure exploitation, finding global solutions to difficult nonconvex MINLP problems will remain out of reach for many applications.

Keywords Mixed-integer nonlinear programming · Global optimization · Software

Mathematics Subject Classification 65K05 · 90C11 · 90C30 · 90C26

✉ Ashutosh Mahajan
amahajan@iitb.ac.in

Sven Leyffer
leyffer@mcs.anl.gov

Jeff Linderoth
linderoth@wisc.edu

James Luedtke
jim.luedtke@wisc.edu

Todd Munson
tmunson@mcs.anl.gov

1 Industrial Engineering and Operations Research, IIT Bombay, Mumbai 400076, India

2 Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL 60439, USA

3 Department of Industrial and Systems Engineering, University of Wisconsin-Madison, Madison, WI 53706, USA

1 Introduction, background, and motivation

Over the past two decades, mixed-integer nonlinear programming (MINLP) has emerged as a powerful modeling paradigm that arises in a broad range of scientific, engineering, and financial applications; see, e.g., [7,28,41,57,66,71]. MINLP combines the combinatorial complexity of discrete decision variables with the challenges of nonlinear expressions, resulting in a class of difficult nonconvex optimization problems. The nonconvexities can arise from both the integrality restrictions and nonlinear expressions. MINLP problems can be generically expressed as

$$\begin{cases} \underset{x}{\text{minimize}} & f(x), \\ \text{subject to} & c(x) \le 0, \\ & x \in \mathcal{X},\ x_i \in \mathbb{Z},\ \forall i \in \mathcal{I}, \end{cases} \tag{1.1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and $c : \mathbb{R}^n \to \mathbb{R}^m$ are given functions, $\mathcal{X} \subset \mathbb{R}^n$ is a bounded polyhedral set, and $\mathcal{I} \subseteq \{1, \ldots, n\}$ is the index set of the integer variables. Equality and range constraints can be readily included in (1.1).

MINLP problems are at least NP-hard combinatorial optimization problems because they include mixed-integer linear programming (MILP) as a special case [50]. In addition, general nonconvex MINLP problems can be undecidable [49]. In the remainder of this paper, we consider only MINLP problems (1.1) that are decidable by assuming that either $\mathcal{X}$ is compact or the problem functions, f and c, are convex. In reality, the distinction between hard and easy problems in MINLP is far more subtle, and instances of NP-hard problems are routinely solved by state-of-the-art solvers.

Figure 1 provides an overview of the problem classes within the generic MINLP formulation in (1.1). At the top level, we divide MINLP problems into convex and nonconvex problems (green arrows), where convex refers to problems in which the function defined in the nonlinear constraints and objective are convex. Next, we further divide the problem classes depending on whether they contain discrete variables or not (red arrows). Then we subdivide the problems further by the class of functions that are present. Figure 1 illustrates the broad structural diversity of MINLP. In addition to the standard problem classes of nonlinear programming (NLP), quadratic programming (QP), linear programming (LP), and their mixed-integer (MI) versions, our tree includes second-order cone programming (SOCP) and polynomial optimization, which have received much interest [52–54]. This tree motivates the development of a flexible software framework for specifying and solving MINLP problems that can be extended to tackle different classes of constraints.

Existing solvers for convex MINLP problems include α-ECP [77], BONMIN [9], DICOPT [75], FilMINT [1], GuRoBi [46] (for convex MIQP problems with quadratic constraints), KNITRO [14,79], MILANO [8], MINLPBB [55], and SBB [12]. These solvers require only first and second derivatives of the objective function and constraints. The user can either provide routines that evaluate the functions and their derivatives at given points or use modeling tools such as AMPL [30], GAMS [11], or Pyomo [47] to provide them automatically. These solvers are not designed to exploit

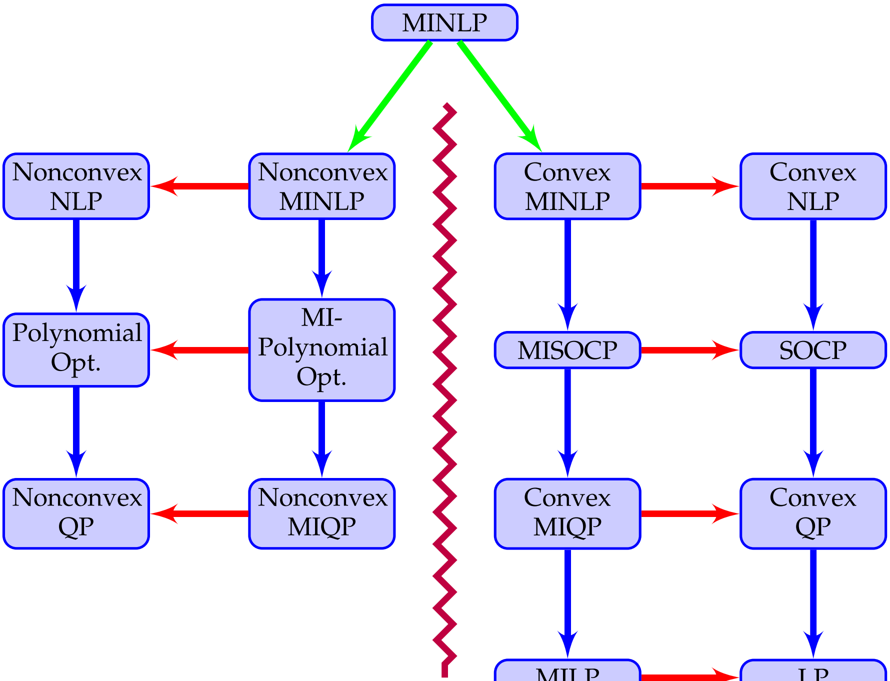

Fig. 1 MINLP problem class tree (color figure online)

the structure of nonlinear functions. While the solvers can be applied to nonconvex MINLP problems, they are not guaranteed to find an optimal solution.

On the other hand, existing solvers for nonconvex MINLP include α-BB [4], ANTIGONE [62], BARON [67], COCONUT [63,69], Couenne [6], CPLEX [48] (for MIQP problems with a nonconvex objective function), GloMIQO [61], and SCIP [2,74]. These solvers require the user to explicitly provide the definition of the objective function and constraints. While linear and quadratic functions can be represented by using data stored in vectors and matrices, other nonlinear functions are usually represented by means of computational graphs. A computational graph is a directed acyclic graph (DAG). A node in the DAG represents either a variable, a constant, or an operation (e.g., $+$, $-$, $\times$, $/$, exp, log). An arc connects an operator to its operands. An example of a DAG is shown in Fig. 2. Functions that can be represented by using DAGs are referred to as "factorable functions." Modeling languages allow the user to define nonlinear functions in a natural algebraic form, convert these expressions into DAGs, and provide interfaces to read and copy the DAGs.

Algorithmic advances over the past decade have often exploited special problem structure. To exploit these advances, MINLP solvers must be tailored to special problem classes and nonconvex structures. In particular, a solver must be able to evaluate, examine, and possibly modify the nonlinear functions. In addition, a single MINLP solver may require several classes of relaxations or approximations to be solved as subproblems, including LP, QP, NLP, or MILP problems. For example, our QP-diving

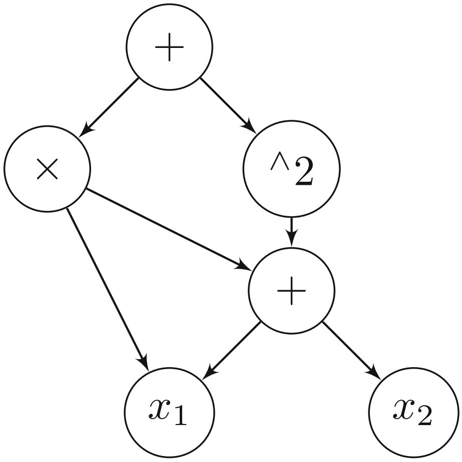

Fig. 2 A directed acyclic graph representing the nonlinear function $x_1(x_1 + x_2) + (x_1 + x_2)^2$

approach [58] solves QP approximations and NLP relaxations. Different nonconvex structures benefit from tailored branching, bound tightening, cut-generation, and separation routines. In general, nonconvex forms are more challenging and diverse than integer variables, thus motivating a more tailored approach. Moreover, the emergence of new classes of MINLP problems such as MILP problems with second-order cone constraints [20,21] and MINLP problems with partial-differential equation constraints [56] necessitates novel approaches.

These challenges and opportunities motivate the development of our Minotaur software framework for MINLP. Minotaur stands for Mixed-Integer Nonlinear Optimization Toolkit: Algorithms, Underestimators, and Relaxations. Our vision is to enable researchers to implement new algorithms that take advantage of problem structure by providing a general framework that is agnostic of problem type or solvers. Therefore, the goals of Minotaur are to (1) provide reliable, efficient, and usable MINLP solvers; (2) implement a range of algorithms in a common framework; (3) provide flexibility for developing new algorithms that can exploit special problem structure; and (4) reduce the burden of developing new algorithms by providing a common software infrastructure.

The remainder of this paper is organized as follows. In Sect. 2, we briefly review some fundamental algorithms for MINLP and highlight the main computational and algorithmic components that motivate the design of Minotaur. In Sect. 3, we describe Minotaur's class structure and introduce the basic building blocks of Minotaur. In Sect. 4, we show how we can use this class structure to implement the basic algorithms described in Sect. 2. Section 5 presents some extensions to these basic algorithms that exploit additional problem structure, including a nonlinear presolve and perspective reformulations. This section illustrates how one can take advantage of our software infrastructure to build more complex solvers. Section 6 summarizes our conclusions. Throughout, we demonstrate the impact of our techniques on sets of benchmark test problems and show that we do not observe decreased performance for increased generality.

2 General algorithmic framework

Minotaur is designed to implement a broad range of relaxation-based tree-search algorithms for solving MINLP problems. In this section, we describe the general algorithmic framework and demonstrate how several algorithms for solving MINLP problems fit into this framework. We concentrate on describing single-tree methods for *convex MINLP problems* (i.e., MINLP problems for which the nonlinear functions f and c are convex), such as nonlinear branch-and-bound [18,45,51] and LP/NLP-based branch-and-bound [65]. We also describe how nonconvex MINLP problems fit into our framework. We do not discuss multitree methods such as Benders decomposition [38,72], outer approximation [9,22,25], or the extended cutting plane method [70,78], although these methods can be implemented by extending some of the source code of Minotaur.

2.1 Relaxation-based tree-search framework

The pseudocode in Algorithm 1 describes a basic tree-search algorithm. In the algorithm, P, P', and Q represent subproblems of the form (1.1), which may be obtained by reformulating a problem or by adding restrictions introduced in the branching phase. The set $\mathcal{O}$ maintains a list of open subproblems that need to be processed. When this list is empty, the algorithm terminates. Otherwise, a node is selected from the list and "processed." The results from processing a node are used to (1) determine whether a new feasible solution is found and, if it is better than an existing solution, to update the *incumbent* solution and (2) determine whether the subproblem can be pruned. If the subproblem cannot be pruned, then the subproblem is *branched* to obtain additional subproblems that are added to $\mathcal{O}$.

```
Algorithm: Tree Search
Reformulate P to obtain problem P'.
Place P' in the set O.
repeat
    Node selection: Select and remove a problem Q from O
    (FeasSolutionFound,CanPrune) ← Process node Q
    if FeasSolutionFound then
        Update incumbent
    else if not CanPrune then
        Branch: Create subproblems and add to O
until O is empty
```

Algorithm 1: Generic tree-search algorithm for solving a MINLP problem P.

The standard mechanism in Minotaur for node processing is described in Algorithm 2, which constructs relaxations of the subproblem Q. A relaxation of Q is a problem R such that the optimal value of R (when minimizing) is guaranteed to be a lower bound on the optimal value of Q. After the relaxation is constructed, the relaxation problem is solved to obtain its optimal value. If the relaxation problem is infeasible or if its optimal value matches or exceeds the value of a known upper bound, then the subproblem can be pruned. Otherwise, the relaxation may be updated (this step might be skipped in some algorithms); and, if the relaxation solution x^R is no longer feasible, then the updated relaxation is solved.

The relaxation used is a key characteristic of a tree-search algorithm. The basic requirements for a relaxation are that it provides a lower bound on the optimal value of a subproblem and can be solved by an available solver. Tighter relaxations are preferred because they typically result in smaller branch-and-bound search trees. Creating and updating (refining) relaxations from the description of subproblem Q are critical computational tasks.

```
Algorithm: Process Node Q
Input: Incumbent value v^I and subproblem Q
Output: FeasSolutionFound, CanPrune, and possibly updated incumbent value v^I
FeasSolutionFound ← FALSE, CanPrune ←FALSE
Construct a relaxation R of Q
repeat
    Solve relaxation: Solve problem R to obtain value v^R and solution x^R
    if R is infeasible or v^R ≥ v^I then
        CanPrune ← TRUE
    else if x^R is feasible for Q then
        FeasSolutionFound ← TRUE
    else
        Update Relaxation R
until CanPrune or FeasSolutionFound or x^R feasible for updated R
```

Algorithm 2: Generic relaxation-based algorithm for processing a node Q.

Minotaur provides a basic infrastructure for managing and storing the open nodes in the tree-search algorithm (the *tree*), for interfacing to modeling languages and subproblems solvers, and for performing basic housekeeping tasks, such as timing and statistics. Section 3 shows how these computational components are implemented in Minotaur's class structure. The remaining subsections illustrate how these components are used to build standard solvers and to develop more advanced MINLP solvers.

2.2 Nonlinear branch and bound for convex MINLPs

In nonlinear branch and bound (NLPBB) for convex MINLP problems, a relaxation R of a subproblem Q is obtained by relaxing the constraints $x_j \in \mathbb{Z}$ to $x_j \in \mathbb{R}$ for all $j \in I$. The resulting problem is a continuous NLP problem; and when all functions defining the (reformulated) MINLP P' are smooth and convex, Q can be solved to global optimality with standard NLP solvers.

If the solution of the relaxation x^R is integer feasible (i.e., $x_j^R \in \mathbb{Z}$ for all $j \in I$), then the relaxation solution is feasible and the node processor sets variable `FeasSolutionFound` to TRUE. If the relaxation is infeasible or its optimal value is at least as large as the incumbent optimal value, then the subproblem can be pruned. Otherwise, branching must be performed. In this case, branching is performed by choosing a variable x_j with $j \in I$ such that $x_j^R \notin \mathbb{Z}$. Then, two new subproblems are created by adding new bounds $x_j \leq \lfloor x_j^R \rfloor$ and $x_j \geq \lceil x_j^R \rceil$, respectively, and these subproblems are added to $\mathcal{O}$.

2.3 LP/NLP-based branch-and-bound algorithms for convex MINLPs

Modern implementations of the LP/NLP-based branch-and-bound method [65] are among the most powerful solvers [1,9] for convex MINLP problems. The basic idea is to replace the NLP relaxation used in NLPBB with an LP relaxation. This LP relaxation is constructed by relaxing the constraints $x_j \in \mathbb{Z}$ to $x_j \in \mathbb{R}$ for all $j \in I$ and by replacing the nonlinear functions f and c with piecewise-linear lower bounds obtained from first-order Taylor-series approximations about a set of points $x^{(l)}$ for $l \in \mathcal{L}$. The convexity of the problem functions ensures that this linearization provides an outer approximation. As usual, if this relaxation is infeasible or its objective value is at least as large as the incumbent objective value, then the subproblem can be pruned.

Feasibility of the relaxation solution x^R is checked with respect to both the integrality constraints and the relaxed nonlinear constraints.

1. If x^R is feasible for both, then the incumbent is updated, and the node is pruned.
2. If x^R is integer feasible, but violates a nonlinear constraint, then the relaxation is updated by fixing the integer variables $x_j = x_j^R$ for all $j \in I$ and solving the resulting continuous NLP subproblem. If the NLP subproblem is feasible and improves upon the best-known solution, then the incumbent is updated. Whether the NLP subproblem is feasible or not, the set of linearization points $x^{(l)}$ for $l \in \mathcal{L}$ is updated so that the LP relaxation is refined.
3. If x^R is not integer feasible, then either the LP relaxation can be refined (e.g., by updating the set of linearization points so that the relaxation solution x^R is no longer feasible), or we can choose to exit the node processing.

If the node processing terminates with a relaxed solution that is not integer feasible, then, as in NLPBB, the subproblem is subdivided by choosing an integer variable j with $x_j^R \notin \mathbb{Z}$ and updating the bounds in the two subproblems.

2.4 Branch and bound for nonconvex MINLPs

If the problem functions f or c are nonconvex, then standard NLP solvers are not guaranteed to solve the continuous relaxation of (1.1) to global optimality. In order to ensure that the relaxations remain solvable, convex relaxations of the nonconvex feasible set must be created. In such relaxations, the quality of the outer approximation depends on the tightness of the variable bounds. The details of such a relaxation scheme for nonconvex quadratically constrained quadratic programs are described in Sect. 4. A key difference is that in addition to branching on integer variables, this algorithm requires branching on continuous variables that appear in nonconvex expressions in (1.1). Thus, in the branching step, subproblems may be created by subdividing the domain of a continuous variable. The updated lower and upper bounds are then used when these subproblems are processed to obtain tighter convex outer approximations of the nonconvex feasible region.

3 Software classes in Minotaur

The Minotaur framework is written in C++ by using a class structure that allows developers to easily implement new functionality and exploit structure. By following a modular approach, the components remain interoperable and compatible if the functions they implement are compatible. Thus, developers can customize only a few selected components and use the other remaining components to produce new solvers. In particular, a developer can override the default implementation of only a few specific functions of a class by creating a "derived C++ class" that implements these functions using methods or algorithms different from the base class. This approach also facilitates easy development of extensions to solvers, such as MINLP solvers with nonconvex nonlinear expressions.

Our framework has three main parts: (1) core, (2) engine, and (3) interface. The core includes all methods and data structures used while solving a problem, for example, those to store, modify, analyze, and presolve problems, create relaxations and add cuts, and implement the tree search and heuristic searches. The engine includes routines that call various external solvers for LP, QP, or NLP relaxations or approximations. The interface contains routines that read input files in different formats and construct an instance. We first describe the most commonly used classes in these three parts and then demonstrate how some can be overridden.

3.1 `Core`

The C++ classes in the core can be classified into four types based on their use.

3.1.1 Classes used to represent and modify a problem

A MINLP problem is represented by means of the `Problem` class. This class stores pointers to all the variables and constraints and the objective and provides methods to query and modify them. Each variable is an object of the `Variable` class. Similarly, constraints and the objective function are objects of the `Constraint` and the `Objective` classes, respectively. A separate `SOS` class is provided for storing special ordered sets (e.g., SOS-1 and SOS-2). This scheme provides a natural and intuitive representation of the problem that is easy to modify. Table 1 lists the main classes used in defining a MINLP problem in Minotaur and their brief description.

Since the objective and constraints of the MINLP problem may have general nonlinear functions, we require specialized data structures for these functions. The `Constraint` and `Objective` classes store a pointer to an object of the `Function` class. The `Function` class in turn has pointers to objects of the `LinearFunction`, `QuadraticFunction`, and `NonlinearFunction` classes and provides other operations. Thus, we store the mathematical function of a constraint or objective as

Table 1 Base classes used in defining a MINLP problem

Name of class	Description
`Variable`	Store a representation of a variable including its index in the instance, its type (integer, binary or continuous), its lower and upper bounds, and other useful information, such as the list of constraints where it appears and whether it appears in any nonlinear functions
`Function`	Store a mathematical function of the variables and define C++ functions to evaluate the function, gradient, and Hessian at a given point. It also has routines to infer bounds on function values, check the type of function, add another mathematical function, replace a variable by another one, remove variables, and implement other tasks
`LinearFunction`	Store a linear function of variables and implements methods to evaluate it at a given point and to query and modify the coefficients
`QuadraticFunction`	Store a quadratic function of variables and implements methods to evaluate the function, gradient, and Hessian at a given point and query and modify the coefficients
`NonlinearFunction`	Store a nonlinear function of variables and implements methods to evaluate the function, gradient, and Hessian at a given point
`Constraint`	Store and modify properties of a constraint including its index in the problem, its lower and upper bounds, and the functions used by the constraint
`Objective`	Store and modify properties of the objective including the functions used by the objective, its sense (maximize or minimize), and the constant offset
`Problem`	Store a representation of a MINLP problem. The object stores `Variable`, `Constraint`, and `Objective` objects. Additionally it has routines to store and query the Jacobian and Hessian of the Lagrangian and other problem-specific objects, such as `SOS` variables

a sum of a linear component, a quadratic component, and a general nonlinear component. A linear constraint, for example, is represented by a `Constraint` object whose `Function` class has a pointer to only a `LinearFunction`; the pointers to `QuadraticFunction` and `NonlinearFunction` are null. The `NonlinearFunction` class has several derived classes that we describe next.

The `CGraph` Class is a derived class of the `NonlinearFunction` class used to store any factorable function. As described in Sect. 1, it stores a nonlinear function in the form of a directed acyclic graph. Each node of DAG is assumed to be scalar valued. The DAG is stored as a vector of objects of class `CNode`. Each `CNode` object represents either an operator ($+$, $-$, $|.|$,etc), a constant number or a variable of the `Problem`. Each `CNode` except the one corresponding to the output (topmost node) of the DAG has at least one parent and also contains pointers to its child and parent `CNode`-objects. Table 2 lists the operators supported by `CGraph`.

`CGraph` class has DAG-specific methods, such as adding or deleting a node or changing a variable or a constant. These methods can be used to create and modify any factorable function by using a given set of operators. For instance, Fig. 3 shows an excerpt of code that can be used to create an object of `CGraph` class corresponding to the example DAG from Fig. 2. A more complicated example is shown in Fig. 4 that

Table 2 List of operators supported by `CGraph` class

Name	Operation	Name	Operation	Name	Operation
OpAbs	$\lvert x\rvert$	OpDiv	x/y	OpRound	$\lfloor x \rceil$
OpAcos	$\arccos(x)$	OpExp	e^x	OpSin	$\sin(x)$
OpAcosh	$arccosh(x)$	OpFloor	$\lfloor x \rfloor$	OpSinh	$\sinh(x)$
OpAsin	$\arcsin(x)$	OpLog	$\ln(x)$	OpSqr	x^2
OpAsinh	$arcsinh(x)$	OpLog10	$\log_{10}(x)$	OpSqrt	$\sqrt{x}$
OpAtan	$\arctan(x)$	OpMinus	$x - y$	OpSumList	$\sum_{i=1}^{n} x_i$
OpAtanh	$arctanh(x)$	OpMult	$x \times y$	OpTan	$\tan(x)$
OpCeil	$\lceil x \rceil$	OpNum		OpTanh	$\tanh(x)$
OpCos	$\cos(x)$	OpPlus	$x + y$	OpUMinus	$-x$
OpCosh	$\cosh(x)$	OpPow	x^y	OpVar	x
OpCPow	x^k	OpPowK	x^k		

x, y denote the operands. Only OpSumList accepts more than two operands

constructs the function needed for an approximation of the perspective formulation of a given nonlinear expression; see Sect. 5.

Being a derived class of `NonlinearFunction`, the `CGraph` class also contains routines for evaluating the gradient and Hessian of the function it stores. We have implemented automatic differentiation techniques [15,36,40] for these purposes. In particular, the gradient is evaluated by using reverse mode. The Hessian evaluation of a function $f : \mathbb{R}^n \to \mathbb{R}$ uses at most n evaluations, one for each column of the Hessian matrix. In each iteration, say i, we first evaluate $\nabla f(x)^T e_i$ in a forward-mode traversal and then perform a reverse-mode traversal to compute the ith column of the Hessian (see, e.g., [64, Ch. 7]). Exploiting sparsity for faster evaluation of Hessian [37] is currently not implemented, but will be tried in the future. The call to the derivative evaluation returns an error if the `CGraph` object has an operator that does not permit differentiation (like `OpAbs`) or if the derivative is not defined at the given point (e.g. $\sqrt{x}$ at $x = 0$).

Besides computing the derivatives, the `CGraph` class is used for finding bounds on the values that the nonlinear function can assume over the given ranges of variables. Conversely, it can deduce bounds on the values that a variable can assume from given lower or upper bounds on the nonlinear function and other variables. These techniques are called feasibility-based bound tightening [6].

The `MonomialFunction` *Class* is a derived class of the `NonlinearFunction` class used for representing monomial functions of the form $a \prod_{i \in J} x_i^{p_i}$, where $a \in \mathbb{R}$, $p_i \in \mathbb{Z}_+, i \in J$, and the set J are given. This class stores the pointer to the variables and the powers in a C++ map data structure.

The `PolynomialFunction` *Class* is a derived class of the `NonlinearFunction` class used for representing polynomial functions. It keeps a vector of objects of the `MonomialFunction` class to represent the polynomial.

```
ProblemPtr p = (ProblemPtr) new Problem();
VariablePtr x1 = p->newVariable(0,1,Binary);
VariablePtr x2 = p->newVariable(0,1,Binary);

CGraphPtr cg = (CGraphPtr) new CGraph();
CNode *nx1 = cg->newNode(x1);
CNode *nx2 = cg->newNode(x2);
CNode *np  = cg->newNode(OpPlus, nx1, nx2);
CNode *nm  = cg->newNode(OpMult, nx1, np);
CNode *ns  = cg->newNode(OpSqr, np);

cg->setOut(cg->newNode(OpPlus, nm, ns));
cg->finalize(); cg->write(std::cout);
```

Fig. 3 Excerpt of code used to create and display the nonlinear function in two variables $x_1(x_1 + x_2) + (x_1 + x_2)^2$

Table 3 Base classes used in implementing branch-and-bound algorithms for solving MINLP problems

Name of class	Description
`Brancher`	Determine how to branch a node of the branch-and-bound tree to create new subproblems
`Engine`	Solve LP, QP, NLP, or some other problem using an external solver such as FilterSQP, IPOPT, or CLP
`Node`	Store information about a node of the branch-and-bound tree such as the lower bound on the optimal value, the branching variable (or object), and pointers to child and parent nodes
`NodeRelaxer`	Create or modify a relaxation of the problem at a given node in the branch-and-bound tree
`TreeManager`	Store, add, and delete nodes of the branch-and-bound tree and select the node for processing
`ActiveNodeStore`	Store active or open nodes of the branch-and-bound tree in an appropriate data structure and compute the global lower bound from the bounds of the individual nodes
`CutManager`	Store, add, and remove cuts from the problem
`NodeProcessor`	Process a node of the branch-and-bound tree by solving the relaxation and/or adding cuts, presolving, branching, or applying heuristics

3.1.2 Classes used in branch-and-bound algorithms

To keep the design of branch-and-bound algorithms modular and flexible, a base class is defined for every step of the algorithm described in Sect. 2. Table 3 lists some of the main classes and their functionality. The user can derive a new class for any of these steps without modifying the others.

We illustrate the design principle by means of the `NodeProcessor` class. The class implements the methods `processRootNode()` and `process()`, and other helper functions. Figure 5 depicts two ways of processing a node. In a simple NLPBB solver for convex MINLP problems, we may only need to solve an NLP relaxation of the problem at the current node. This procedure is implemented in the

```
CGraph* PerspRefUT::getPerspRec(CGraph* f, VariablePtr y, double eps,
                                 int *err)
{
  CNode *ynode = 0, *anode = 0;
  CGraph* p = new CGraph();

  ynode = p->newNode(y);
  anode = p->newNode(1.0-eps);
  ynode = p->newNode(OpMult, anode, ynode); // y*(1-eps)
  anode = p->newNode(eps);
  ynode = p->newNode(OpPlus, anode, ynode); // eps + y*(1-eps)

  anode = getPerspRec_(p, f->getOut(), ynode, f); // start recursion

  anode = p->newNode(OpMult, anode, ynode); // final multiplication
  p->setOut(anode); p->finalize();
  return p;
}

CNode* PerspRefUT::getPerspRec_(CGraph* p, const CNode *node,
                                 CNode *znode, CGraph* f)
{
  CNode *newl  = 0, *newr = 0;
  if (OpVar == node->getOp()) {
    newl = p->newNode(f->getVar(node));
    return (p->newNode(OpDiv, newl, znode));
// x/(y(1-eps)+eps)
  } else if (OpNum == node->getOp()) {
    return (p->newNode(node->getVal()));
  } else if (1 == node->numChild()) {
    newl = getPerspRec_(p, node->getL(), znode, f);    // recurse
    return (p->newNode(node->getOp(), newl, newl));
  } else if (2 == node->numChild()) {
    newl = getPerspRec_(p, node->getL(), znode, f);    // recurse
    newr = getPerspRec_(p, node->getR(), znode, f);    // recurse
    return (p->newNode(node->getOp(), newl, newr));
  } else if (2 < node->numChild()) {
    CNode **childrn = new CNode*[node->numChild()];
// array of children
    CNode** c1 = node->getListL();
    CNode** c2 = node->getListR();
    int i = 0;
    while(c1 < c2) {
      childrn[i] = getPerspRec_(p, *c1, znode, f);     // recurse
    }
    return (p->newNode(node->getOp(), childrn, node->numChild()));
  }
  return 0;
}
```

Fig. 4 Excerpt of code used to obtain a CGraph of $p(x, y) = (y(1-\epsilon)+\epsilon)f\left(\frac{x}{y(1-\epsilon)+\epsilon}\right)$ from a given CGraph of $f(x)$ by recursively traversing it

`BndProcessor` class derived from the `NodeProcessor` class. Based on the solution of this NLP, we may either prune the node or branch. For other algorithms, we may need a more sophisticated node processor that can call a cut-generator to add cuts or invoke presolve. The `PCBProcessor` (Presolve, Cut and Bound Processor)

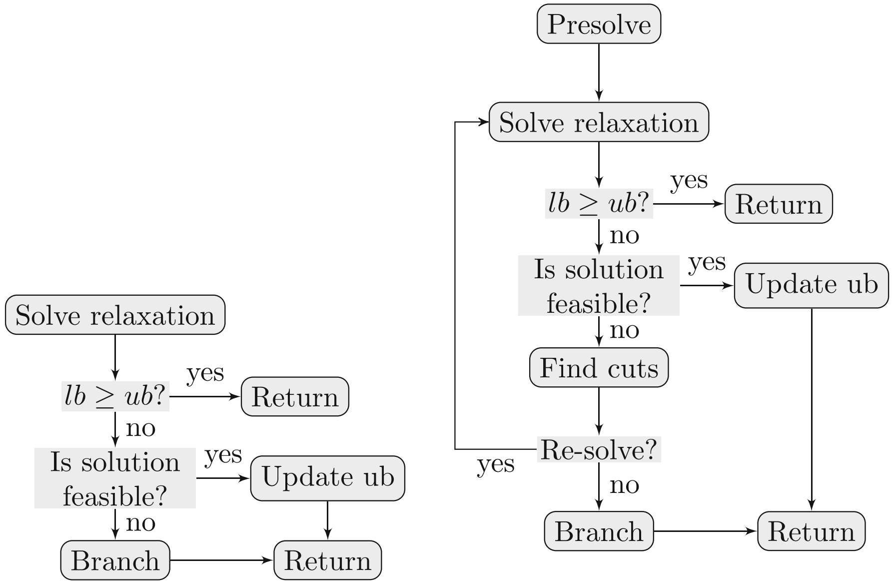

Fig. 5 Two ways of implementing a `NodeProcessor` class in the branch-and-bound method. Left: standard NLPBB; right: LP/NLP-based branch and bound

derived class implements this scheme. The right hand side picture in Fig. 5 represents the scheme implemented in `PCBProcessor`.

Modularity enables us to select at runtime the implementation of each component of the branch-and-bound algorithm. For instance, depending on the we are solving, we can select at runtime one of five branchers: `ReliabilityBrancher`, for implementing reliability branching [3]; `MaxVioBrancher`, for selecting the maximum violation of the nonconvex constraint; `LexicoBrancher`, for selecting the variable with the smallest index; `MaxFreqBrancher`, for selecting a variable that appears as a candidate most often; and `RandomBrancher`, for selecting a random variable. Minotaur also has branchers for global optimization that select continuous variables in order to refine the convex relaxations.

3.1.3 Classes used to exploit problem structure

The classes mentioned in the preceeding sections implement general methods of a branch-and-bound algorithm and do not explicitly exploit the "structure" or "specific difficulties" of a problem. By structure, we mean those special properties of the constraints and variables that can be exploited (and, sometimes, must be exploited) to solve the problem. For instance, if we have integer decision variables and linear constraints, then we can generate valid inequalities to make the relaxation tighter by using the techniques developed by the MILP community. Similarly, if some constraints of the problem are nonlinear but convex, then we can create their linear relaxation by deriving linear underestimators from a first-order Taylor approximation.

Table 4 Main functions of a `Handler`

Name of the function	Description
`relaxNodeFull`	Create a new relaxation of the constraints being handled
`relaxNodeInc`	Create a relaxation by adding relevant modifications to the relaxation obtained at the parent node
`presolve`	Presolve the specific structure
`presolveNode`	Presolve the specific structure at a given node
`isFeasible`	Check whether a point is feasible to the constraints being handled
`separate`	Store, add, and remove valid inequalities for the specific structure
`getBranchingCandidates`	Shortlist candidates for branching in a node of the branch-and-bound tree
`branch`	Create branches of the current node using the branching candidate shortlisted by this handler and selected by the brancher

We use a `Handler` class to enable implementation of routines that exploit specific problem structures. This idea is inspired from the "Constraint Handlers" used in SCIP [2] that ensure that a solution satisfies all the constraints of a problem. Since special properties or structures can be exploited in a branch-and-bound algorithm at many different steps, `Handler` instances are invoked at all these steps: creating a relaxation, presolving, finding valid inequalities, checking whether a given point is feasible, shortlisting candidates for branching, and creating branches. Table 4 lists some of the important functions of a `Handler`.

The `Handler` base class in Minotaur is abstract; it declares only the functions that every `Handler` instance must have and leaves the implementation of structure-specific methods to the developer. We describe a few commonly used `Handler` types implemented for our solvers.

`IntVarHandler` is one of the simplest handlers. It ensures that a candidate accepted as a solution satisfies all integer constraints at a given point. It implements only three of the functions listed in Table 4: `branch`, `isFeasible`, and `getBranchingCandidates`. The first function checks whether the value of each integer-constrained variable is integer within a specified tolerance. The second function returns a list of all the integer variables that do not satisfy their integer constraints. The last function creates two branches of a node if the `Brancher` selects an integer variable for branching.

`SOS2Handler` is used when the MINLP problem contains SOS-2 variables [5]. Its `isFeasible` routine checks whether at most two consecutive variables in the set are nonzero. The `getBranchingCandidates` routine returns two subsets, one for each branch. In the first branch all variables of the first subset are fixed to zero, and in the second all variables of the second subset are fixed to zero.

`QGHandler` is a more complicated handler used when solving convex MINLP problems with the LP/NLP-based algorithm of Quesada and Grossmann [65] (QG

stands Quesada-Grossmann.) This handler does not implement any `branch` or `getBranchingCandidates` routines. The `isFeasible` routine checks whether the nonlinear constraints are satisfied by the candidate solution. If the problem is not feasible, then the `separate` routine solves a continuous NLP relaxation and obtains linear inequalities that cut off the given candidate. These valid inequalities are added to the LP relaxation, which is then solved by the `NodeProcessor`.

`NlPresHandler` is for applying presolving techniques to nonlinear functions in the constraints and objective. Therefore, it implements only the `presolve` and `presolveNode` functions. This handler tries to compute tighter bounds on the variables in nonlinear constraints by performing feasibility-based bound tightening on one constraint at a time. It also checks whether a nonlinear constraint is redundant or infeasible.

`QuadHandler` is for constraints of the form

$$x_i^2 = x_k, \quad \text{or} \tag{3.1}$$

$$x_i x_j = x_k \tag{3.2}$$

for some $i, j, k \in \{1, \dots, n\}$, with bounds on all variables given by $l_i \le x_i \le u_i$ for all $i \in \{1, \dots, n\}$. This handler ensures that a solution satisfies all constraints of type (3.1) and (3.2) of a MINLP problem. It implements all the functions in Table 4. The `RelaxNodeFull` and `RelaxNodeInc` functions create linear relaxations of these constraints by means of the McCormick inequalities [60],

$$(l_i + u_i)x_i - x_k \ge l_i u_i \tag{3.3}$$

for constraint (3.1) and

$$\begin{aligned} l_j x_i + l_i x_j - x_k &\le l_i l_j \\ u_j x_i + u_i x_j - x_k &\le u_i u_j \\ l_j x_i + u_i x_j - x_k &\ge u_i l_j \\ u_j x_i + l_i x_j - x_k &\ge l_i u_j \end{aligned} \tag{3.4}$$

for the bilinear constraint (3.2). The `presolve` and `nodePresolve` routines find tighter bounds, $l_i \le u_i$, on the variables based on the bounds of the other two variables. A lower bound on x_k, for example, is $\min\{l_i l_j, u_i l_j, l_i u_j, u_i u_j\}$. The `isFeasible` routine checks whether a given candidate solution satisfies all the constraints of the form (3.1) and (3.2). If the given point $\hat{x}$ has $\hat{x}_i^2 > \hat{x}_k$, then the `separate` routine generates a valid inequality

$$2\hat{x}_i x_i - x_k \le \hat{x}_i^2.$$

If $\hat{x}_i^2 < \hat{x}_k$ for constraint (3.1) or $\hat{x}_i \hat{x}_j \ne \hat{x}_k$ for constraint (3.2), then the `getBranchingCandidates` routine returns x_i and x_j as candidates for branching. If one of these is selected for branching by the `Brancher`, then the `branch` routine creates two branches by modifying the bounds on the branching variable.

3.1.4 Transformer

It is frequently the case that the specific structure a handler supports, such as the quadratic constraints (3.1) and (3.2), may occur in more complicated functions of a given problem formulation. To enable application of handlers for those structures, we first need to transform a problem into a form that is equivalent to the original problem, but exposes these simpler structures. The `Transformer` class performs this task. The default implementation of the `Transformer` traverses the DAG of a nonlinear function in a depth-first search and adds new variables to create simple constraints of the desired form. The process can be better explained with an example. Consider the constraint

$$x_1(x_1 + x_2) + (x_1 + x_2)^2 \leq 1.$$

Its computational graph was earlier shown in Fig. 2. The `Transformer` reformulates it as

$$\begin{aligned} x_3 &= x_1 + x_2, \\ x_4 &= x_1 x_3, \\ x_5 &= x_3^2, \\ x_4 + x_6 &\leq 1, \end{aligned}$$

where we have used unary and binary operations for simplicity. The `Transformer` also maintains a list of new variables it introduces and the corresponding expression they represent. These variables are then reused if the same expression is observed in other constraints or the objective function. In this example, if the expression $x_1 + x_2$ is observed in some other computational graph, then x_3 will be reused. Similarly, x_4 and x_5 will be reused. The code uses a simple hashing function to detect these common subexpressions. In addition to applying the transformations, the `Transformer` assigns every nonlinear constraint to a specific `Handler`. Since there are often many alternative reformulations of a problem, a different implementation of the `Transformer` class may lead to a different reformulation and can have significant impact on algorithm performance.

3.1.5 Utility classes

Utility classes provide commonly required functions such as measuring the CPU time (`Timer` class), writing logs (`Logger`), parsing and storing user-supplied options (`Options`), and commonly used operations on vectors, matrices, and intervals (`Operations`).

3.2 `Engine`

The Minotaur framework calls external libraries for solving relaxations or simpler problems. A user can choose to link the Minotaur libraries with several external libraries. Minotaur libraries and executables can also be compiled without any of these external libraries.

The Open-Solver Interface (OSI) library provided by COIN-OR [17] is used to link to the CLP [29], GuRoBi [46], and CPLEX [48] solvers for LP problems. The BQPD [24] and qpOASES [23] solvers can be used to solve QP problems. FilterSQP [26,27] and IPOPT [76] can be used to solve NLP problems using active-set and interior-point methods, respectively.

The interface to each solver is implemented in a separate class derived from the `Engine` abstract base class. For instance, the `BQPDEngine` class implements an interface with the BQPD solver. The two main functions of an `Engine` class are to (1) convert a problem represented by the `Problem` class to a form required by the particular solver and (2) convert and convey the solution and solution status to the Minotaur routines. The conversion of LP and QP problems is straightforward. The engine sets up the matrices and vectors in the format required by the solver before passing them. More general NLP solvers such as FilterSQP and IPOPT require routines to evaluate derivatives. These engine classes implement callback functions. For instance, the `FilterSQPEngine` class implements the functions `objfun`, `confun`, `gradient`, and `hessian` that the FilterSQP solver requires, and the `IpoptEngine` class implements the `eval_f`, `eval_grad_f`, and `eval_h` functions required by the `IpoptFunInterface` class.

In a branch-and-bound algorithm, these engines are called repeatedly with only minor changes to the problem. Therefore, one can use the solution information from the previous invocation to *warm-start* the next. The `BQPDEngine`, `FilterSQPEngine`, and `IpoptEngine` classes implement methods to save and use the warm-starting information from previous calls. The OSI interface for LP solvers already provides routines to save and load warm-starting information. The LP engines of Minotaur merely use these features and do not implement any routines for saving the information about the basis associated with the last solution.

3.3 Interface

The `Interface` consists of classes that convert MINLP problems written in a modeling language or a software environment to Minotaur's `Problem` class and other associated classes. In the current version of the Minotaur framework, we have an interface only to AMPL. This interface can be used in two modes.

In the first mode, the `AMPLInterface` class reconstructs the full computational graph of each nonlinear function in the problem. The class uses AMPL library functions to parse each graph and then converts it into the form required by the `CGraph` class. Derivatives are provided by the `CGraph` class. Once the problem instance is created, the `AMPLInterface` is no longer required to solve the instance.

In the second mode, we do not store the computational graph of the nonlinear functions. Rather, the `AMPLProblem` class is derived from the `Problem` class and stores linear constraints and objective using the default implementation. Nonlinear constraints are not stored by using the `CGraph` class. Instead, pointers to the nonlinear functions stored by the AMPL library are placed in the `AMPLNonlinearFunction` class derived from the NonlinearFunction class. This class calls methods provided by the AMPL solver library to evaluate the function or its gradient at a given point. In this mode, the AMPLInterface class provides an object of class AMPLJacobian to evaluate the Jacobian of the constraint set and AMPLHessian to evaluate the Hessian of the Lagrangian of the continuous relaxation. This mode is useful when implementing an algorithm that only requires evaluating the values and derivatives of nonlinear functions in the problem.

Computational evaluation of the speed of the two modes is presented later in Sect. 4.4 (see Automatic Differentiation). The interface also implements routines to write solution files so that the user can query the solution and solver status within AMPL.

4 Implementing basic solvers in Minotaur

We now describe how the three algorithms presented in Sect. 2 can be implemented by combining different classes of the Minotaur framework. Our goal here is not to develop the fastest possible solver, but rather to demonstrate that our flexible implementation does not introduce additional computational overhead. These demonstrations are implemented as examples in our source code and are only simplistic versions of the more sophisticated solvers available.

The general approach for implementing a solver is to first read the problem instance by using an Interface. The next step is to initialize the appropriate objects of the base or derived classes of NodeRelaxer, NodeProcessor, Brancher, Engine, and Handler. Then an object of the BranchAndBound class is set using these components, and the solve method of the BranchAndBound object is called to solve the instance. The code for these steps can be written in a C++ main() function that can be compiled and linked with the Minotaur libraries to obtain an executable.

4.1 Nonlinear branch-and-bound

The NLPBB algorithms for convex MINLPs (Sect. 2.2) is the simplest to implement. It needs only one handler: IntVarHandler to check whether the solution of a relaxation satisfies integer constraints and to return a branching candidate if it does not satisfy the constraints. The BndProcessor described in Sect. 3.1.2 can be used as the node processor, since we need only compare the lower bound of the relaxation with the upper bound at each node. It needs a pointer to an Engine to solve the relaxation. We use FilterSQP in this example. Since only the bounds on variables of the relaxation change during the tree search, NodeIncRelaxer is used to update the relaxation at every node by changing the bounds on appropriate variables. We need not create any relaxation since the variables, constraints, and objective are the same as in the original problem. We use ReliabilityBrancher as the brancher, which implements a nonlinear version of reliability branching [3,10]. The pointer to the Engine used for BndProcessor can be used for this brancher as well.

```
int main(int argc, char** argv) {
  EnvPtr env = (EnvPtr) new Environment();
  HandlerVector handlers;
  int err = 0;

  env->startTimer(err); assert(err==0); // Start timer
  env->getOptions()->findBool("use_native_cgraph")->setValue(true);
  AMPLInterface* iface = new AMPLInterface(env, "bnb");
  ProblemPtr p = iface->readInstance(argv[1]); // read the problem
  p->setNativeDer();

  // create branch-and-bound objects
  BranchAndBound *bab = new BranchAndBound(env, p);
  IntVarHandlerPtr v_h = (IntVarHandlerPtr) new IntVarHandler(env, p);
  handlers.push_back(v_h); // only one handler required

  // setup engine
  EnginePtr e = (FilterSQPEnginePtr) new FilterSQPEngine(env);
  ReliabilityBrancherPtr rel_br = (ReliabilityBrancherPtr) new
                                  ReliabilityBrancher(env, handlers);
  rel_br->setEngine(e);

  // node processor and relaxer
  NodeProcessorPtr npr = (BndProcessorPtr)
                         new BndProcessor(env, e, handlers);
  npr->setBrancher(rel_br); bab->setNodeProcessor(npr);
  NodeIncRelaxerPtr nr = (NodeIncRelaxerPtr)
                         new NodeIncRelaxer(env, handlers);
  RelaxationPtr rel = (RelaxationPtr) new Relaxation(p);
  nr->setRelaxation(rel); nr->setEngine(e); nr->setModFlag(false);
  bab->setNodeRelaxer(nr);
  bab->shouldCreateRoot(false);

  bab->solve();
  bab->writeStats(std::cout);
  bab->getSolution()->writePrimal(std::cout);
}
```

Fig. 6 Excerpt of code for implementing our NLPBB algorithm, `simple-bnb`, using the Minotaur framework and the AMPL function, gradient, and Hessian evaluation routines

Figure 6 contains the `main()` function used to implement this solver. Additional lines containing directives to include header files are omitted for brevity. We refer to our NLPBB solver as `simple-bnb`.

4.2 LP/NLP-based branch-and-bound

Implementing the LP/NLP-based branch-and-bound method [65] requires us to solve a LP relaxation that is different from that of the original problem. The `QGHandler` described in Sect. 3.1.3 solves the required NLP to find the point around which linearization constraints are added to the relaxation. `LinearHandler` is used to copy the linear constraints from the original problem to the relaxation. The `NodeProcessor` is required to solve several relaxations at each node if cuts are added. Thus we use the `PCBProcessor` described in Sect. 3.1.2. The remaining setup is similar to that of the NLPBB algorithm. Figure 7 shows the main portion

```
// create branch-and-bound object
BranchAndBound *bab = new BranchAndBound(env, p);
EnginePtr nlp_e = (FilterSQPEnginePtr) new FilterSQPEngine(env);
EnginePtr e = (OsiLPEnginePtr) new OsiLPEngine(env);

// setup handlers
IntVarHandlerPtr v_h = (IntVarHandlerPtr) new IntVarHandler(env, p);
LinearHandlerPtr l_h = (LinearHandlerPtr) new LinearHandler(env, p);
QGHandlerPtr q_h = (QGHandlerPtr) new QGHandler(env, p, nlp_e);
l_h->setModFlags(false, true);
q_h->setModFlags(false, true);
handlers.push_back(v_h);
handlers.push_back(l_h);
handlers.push_back(q_h);

// setup engine for solving relaxations and branching
ReliabilityBrancherPtr rel_br = (ReliabilityBrancherPtr) new
                             ReliabilityBrancher(env, handlers);
rel_br->setEngine(e);

// node processor
NodeProcessorPtr np = (PCBProcessorPtr) new PCBProcessor(env, e,
                                                         handlers);
np->setBrancher(rel_br);
bab->setNodeProcessor(np);

// node relaxer
NodeIncRelaxerPtr nr = (NodeIncRelaxerPtr)
                       new NodeIncRelaxer(env, handlers);
nr->setEngine(e); nr->setModFlag(false);
bab->setNodeRelaxer(nr);
bab->shouldCreateRoot(true);

bab->solve();
```

Fig. 7 Excerpt of code for implementing our LP/NLP-based algorithm, `simple-qg`, using the Minotaur framework

of the code for implementing this solver. We refer to our LP/NLP-based solver as `simple-qg`.

4.3 Global optimization of quadratically constrained problems

A simple implementation of the branch-and-bound method for solving nonconvex quadratically-constrained quadratic programming (QCQP) problems demonstrates how the Minotaur framework can be used for global optimization. A `Transformer` first converts a given QCQP problem to a form where each quadratic term is assigned to a new variable by means of constraints of the form (3.1) and (3.2). The `QuadHandler` creates the linear relaxations of each of these constraints. Other components of the solver are similar to those described earlier: the `LinearHandler` copies all the linear constraints in the problem, the `IntVarHandler` is used to check integrality constraints, the `NodeIncRelaxer` is used to update the relaxation at each node, and the `PCBProcessor` is used for processing each node. Any of the branchers available

```
ProblemPtr p, newp;

// create branch-and-bound object
BranchAndBound *bab = new BranchAndBound(env, p);
EnginePtr nlp_e = (FilterSQPEnginePtr) new FilterSQPEngine(env);
EnginePtr e = (OsiLPEnginePtr) new OsiLPEngine(env);

// Call transformer. It creates the required handlers.
SimpTranPtr trans = (SimpTranPtr) new SimpleTransformer(env, p);
trans->reformulate(newp, handlers, err); assert(0==err);

// presolve
PresolverPtr pres = (PresolverPtr) new Presolver(newp, env, handlers);
pres->solve();

// brancher
ReliabilityBrancherPtr rel_br = (ReliabilityBrancherPtr) new
                                ReliabilityBrancher(env, handlers);
rel_br->setEngine(e);

// node processor and relaxer
NodeProcessorPtr nproc = (PCBProcessorPtr) new PCBProcessor(env, e,
                         handlers);
nproc->setBrancher(rel_br);
bab->setNodeProcessor(nproc);
NodeIncRelaxerPtr nr = (NodeIncRelaxerPtr) new NodeIncRelaxer(env,
                       handlers);
nr->setEngine(e);
nr->setModFlag(false);
bab->setNodeRelaxer(nr);
bab->shouldCreateRoot(true);

// start solving
bab->solve();
```

Fig. 8 Excerpt of code for implementing our simple global optimization algorithm, `simple-glob`, for QCQP problems using the Minotaur framework

in the toolkit can be used for branching. Figure 8 shows the implementation of this solver. We refer to our simple global optimization solver as `simple-glob`.

The `QuadHandler` requires bounds on variables in the quadratic constraints to create a linear relaxation. A `Presolver` is used to obtain bounds on all variables before the relaxation is created. The `Presolver` class, in turn, calls the `presolve` function of each `Handler`. When `QuadHandler` sees infinite bounds on a variable that requires a McCormick approximation (3.4), the handler returns an error and the solver stops with an error message.

4.4 Performance of Minotaur

We now present experimental results of the performance of the algorithms presented in this section. We have divided the results into three parts: (1) using automatic differentiation, (2) solving convex MINLPs and (3) solving nonconvex MINLPs. Before describing the experiments we first describe the limits and tolerances used in them.

Since many of the comparisons make use of performance profiles, they are also briefly explained.

Limits, tolerances and errors Given a constraint of the form $lb \leq h(x) \leq ub$, where h is the constraint function and lb, ub are the given lower and upper bounds. We say a point $\hat{x} \in \mathbb{R}^n$ is feasible to the constraint if

$$\min\{lb - e_a, lb - e_r\,|lb|\} \leq \tilde{h}(\hat{x}) \leq \max\{ub + e_a, ub + e_r\,|ub|\}, \tag{4.1}$$

where e_a and e_r are the absolute and relative tolerance parameters for checking feasibility, and $\tilde{h}(\hat{x})$ is the value of the function h at $\hat{x}$ evaluated on the computer. In general h and $\tilde{h}$ are different because of the imprecise numerical computing observed when using floating point numbers. For the solvers we describe in this paper, both these tolerance parameters are fixed to 10^{-6}. The same tolerances are used for checking whether a variable value lies between the given bounds. The absolute tolerance for deciding whether a floating point number is an integer value is also 10^{-6}. These tolerance values can be overridden by a user through command line options feasAbs_tol, feasRel_tol and int_tol.

We say a point $\hat{x} \in \mathbb{R}^n$ is feasible to a given MINLP if it satisifies all constraints and integer constraints within the above tolerances. If $\hat{x}$ is feasible to a MINLP within given tolerances, it may still violate some constraints if the MINLP is transformed to another equivalent form. We check candidate solutions of optimization problems for feasibility on the reformulated problem (i.e. the problem obtained after presolving and transforming) before accepting them as feasible. Numerical tolerances are also used when a branch-and-bound node is checked for pruning. A node is pruned if

$$lb \geq \min\{ub - e_a, ub - e_r\,|ub|\},$$

where lb is the lower bound computed for the node, ub is the best known upper bound on the optimal value, e_a is the absolute tolerance and e_r the relative tolerance for pruning nodes on the basis of objective value. Both tolerance parameters are set to 10^{-5} by default and can be overridden by using command line options solAbs_tol and solRel_tol.

When a bound on a constraint is not finite, we use the INFINITY and -INFINITY macros provided by `cmath` header in C++ to represent these bounds. Solvers used to solve relaxations have their own definitions of infinity. Filter-SQP, for instance, considers 10^{20} as infinity. We translate the bounds suitably in the respective `Engine` class.

Other numerical issues also arise while implementing the branch-and-bound based algorithm. The subsolvers or the engines (LP or NLP) used to solve relaxations at a node (Algorithm 2) may fail to converge. If the engine is able to return a feasible point in such cases, we continue branch-and-bound by branching on a candidate if one can be found. Otherwise, a warning is issued and the node is pruned. It is also possible, especially in global optimization, that the optimal solution of the relaxation at a node is not feasible for the MINLP, but neither branching nor cutting planes are able to cut it off because of numerical issues. In this case also, a warning is issued and the node is pruned. Detailed analysis of such errors and numerical issues is a subject of a different study that can be pursued in the future.

Extended performance profile Extended performance profiles [59] plot the relative performance of a set of solvers (say, S) on a given set of problem instances (say, P). Suppose that the performance measure of interest is the time taken to solve an instance. Let $t_{p,s}$ be the time taken by solver $s \in S$ to solve problem instance p. The 'performance ratio' is defined as:

$$\hat{r}_{p,s} = \frac{t_{p,s}}{\min\{t_{p,i} \mid i \in S, i \neq s\}}. \tag{4.2}$$

The performance ratio is thus the factor by which solver s is slow in solving instance p as compared to the best amongst S, except s, for instance p. The smaller the ratio, the faster the solver s is for p. A ratio higher than one indicates that solver s is slower than the fastest solver for p by this factor. On the other hand a ratio value of, say, 0.2 indicates that s is faster than all others by a factor of at least five. A distribution function $\hat{\rho}_s$ is defined as follows to depict performance ratios of a solver over all instances:

$$\hat{\rho}_s(\tau) = \frac{\left|\{p \in P \mid \hat{r}_{p,s} \leq \tau\}\right|}{|P|}, \tag{4.3}$$

where $\tau \geq 0$. Thus $\hat{\rho}_s(2)$ is the fraction of instances on which solver s is slow by a factor of 2 or less as compared to best of all other solvers put together. Similarly, $\hat{\rho}_s(0.5)$ is the fraction of instances on which solver s is faster than all other solvers by a factor of two or more. An extended performance profile plots $\hat{\rho}_s(\tau)$ for all solvers, one curve for each solver, as a function of τ.

As the name suggests, extended performance profile is an extension of the performance profile [19]. The performance ratio (4.2) differs slightly in the two, but the two profiles are the same for $\tau \geq 1$. Extended performance profiles provide addtional information on how fast the solver is ($0 \leq \tau < 1$) while the performance profile only shows how slow it is ($1 \leq \tau$) as compared to the fastest possible. While a higher curve in the performance profile and extended performance profile denotes a faster solver, generally speaking, the relative positions of solver profiles may change when the set of solvers changes [39]. Hence, conclusions should be drawn carefully: when more than two solvers are profiled, a pairwise comparison is not feasible.

Experimental setup All experiments were done on a computer with a 2.9MHz Intel Xeon CPU E5-2670 processor and 128GB RAM. Hyperthreading was switched off. A single compute core was used to solve each problem instance within a specified time limit of one hour. Debian-8 Linux was the operating system. Minotaur version 0.2 (git revision b944ef6), is used in all experiments along with IPOPT version 3.12.3, Mumps version 4.10.0, and Clp version 0.107.4. The Minotaur, IPOPT, AMPL Solver Library, and OSI-CLP code was compiled with the Gnu "g++" version 4.9.2, while "gfortran" version 4.9.2 was used for the fortran code including FilterSQP, BQPD, and MUMPS (used by IPOPT). The optimization flag was set to '-O3' for both compilers.

Automatic differentiation We first compare the effect of using our own implementation of automatic differentiation on the performance of NLP engines. We set up an

Table 5 Time taken to solve 100 NLP relaxations

Instance	Time (s)		Instance	Time (s)	
	AMPL	CGraph		AMPL	CGraph
batchs151208m	28.52	29.96	o7_ar5_1	5.34	5.27
clay0304m	1.51	1.68	rsyn0820m04m	44.47	44.88
clay0305h	42.36	41.00	rsyn0830m04h	119.22	119.75
flay05h	5.41	5.24	slay07m	1.75	1.80
flay06m	1.13	1.15	slay09h	10.55	10.82
fo7_2	3.76	3.83	smallinvDAXr5b150-165	0.71	1.43
fo9_ar2_1	8.38	8.26	smallinvSNPr3b020-022	2.85	17.35
m6	2.88	2.87	sssd22-08	2.24	2.20
m7_ar4_1	5.78	5.93	syn30m03m	18.53	18.74
no7_ar3_1	4.97	5.06	syn40m04h	71.18	75.16

Derivatives are obtained from Minotaur's AMPL interface (column "AMPL") and Minotaur's `CGraph` class (column "CGraph")

experiment in which an NLP engine solves a sequence of NLPs that differ only in bounds on variables. This setup closely mimics the use of NLP engines in a MINLP solver while ensuring that other MINLP routines such as heuristics, presolving, and valid inequalities do not affect the observations.

To benchmark the derivatives produced with our native `CGraph` class, we compare its computation time with that of the AMPL interface derivatives. In particular, we modified `simple-bnb` to replace the `Reliability Brancher` with `LexicoBrancher`. `LexicoBrancher` simply selects the candidate for branching with the smallest index from the list of branching candidates. This solver thus spends almost the entire time solving the NLP relaxations. Table 5 reports the time taken to process 100 nodes of branch and bound (i.e., 100 NLPs) with derivatives from both `CGraph` and `AMPLInterface` on selected instances. We observe no significant difference between the two for nearly 80% of the instances. `CGraph` is slower by a factor of 8 or less for QCQP problems that have a dense Hessian of the Lagrangian as in the case of the smallinvDAXrx and smallinvSNPrx instances. The times for these problems could be significantly reduced by extracting and storing the vectors and matrices for the quadratic forms. While speed is important, the main goal of `CGraph` is to enable a user to easily and reliably manipulate nonlinear functions. `CGraph` never failed in evaluating derivatives in all runs, thus demonstrating its reliability in solving MINLPs.

Convex MINLP We now benchmark the performance of our `simple-bnb` solver described in Sect. 4.1. We compare its performance in two settings: one using `CGraph` and the other using `AMPLInterface`. We also compare the results with Bonmin's implementation of the branch and bound method. In all three settings, we use the IPOPT solver (with MUMPS and ATLAS-BLAS). We set a limit of one hour for each solver. Of the 356 convex MINLPs available in the MINLPLib-2 collection [13,73], we consider only the 333 instances that have integer variables.

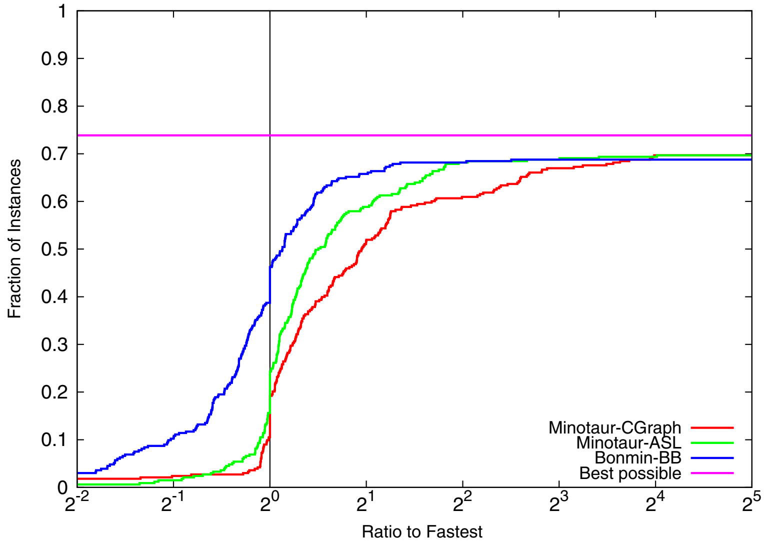

Fig. 9 Extended performance profile based on CPU time for simple NLPBB solvers on 333 convex MINLP problems

The time taken to solve the instances is compared by using the extended performance profiles shown in Fig. 9. The figure shows that our `simple-bnb` method is able to solve almost the same number of instances as is Bonmin (nearly 73% of all instances) within the time limit. On 10% of instances Bonmin is faster than the Minotaur solvers by a factor of 2 or more. This can be seen in the profile by looking at the value for Bonmin at (2^{-1}). The simple implementation with the `AMPLInterface` is slower by a factor of 2 or more as compared with the fastest solver for about 12% of the instances (as Minotaur-ASL value is about 0.6 at 2 and the best possible is 0.73). The same is true for 24% of the instances for `CGraph` and 8% for Bonmin.

Figure 10 compares the performance of our `simple-qg` method described in Sect. 4.2. In this experiment, Bonmin's QG algorithm is selected for solving the problems. The Minotaur solvers use IPOPT for solving NLP problems and OSI-CLP for solving LP problems. The impact of using `CGraph` in this algorithm is much smaller when compared with that of our NLPBB algorithm since fewer NLP relaxations are solved. The performance of the simple implementation is comparable to that of Bonmin even though a gap is discernible in the left half of the graph. This gap is expected because the two examples of Minotaur are nearly identical in behavior and hence only in a few instances is one of them is faster than all other solvers by a large factor.

Global optimization We compare the performance of our `simple-glob` method described in Sect. 4.3 with that of three other solvers: BARON-15.2 (with AMPL interface), Couenne-0.5.6 (with AMPL interface, CLP as the LP solver, IPOPT as the NLP solver, and Cbc as the MILP solver), and SCIP-3.2.0 (with pip interface, SoPlex as the LP solver, and IPOPT as the NLP solver). All solvers were run using default settings. A time limit of one hour was set on each instance. Tests were performed on 266 continuous QP and QCQP problems taken from the MINLPLib-2 collection [13,73].

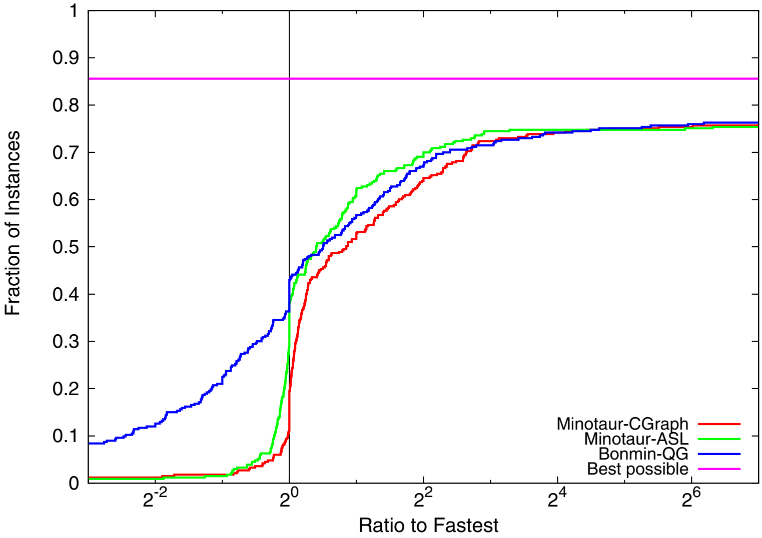

Fig. 10 Extended performance profile based on CPU time of LP/NLP-based solvers on 333 convex MINLP problems

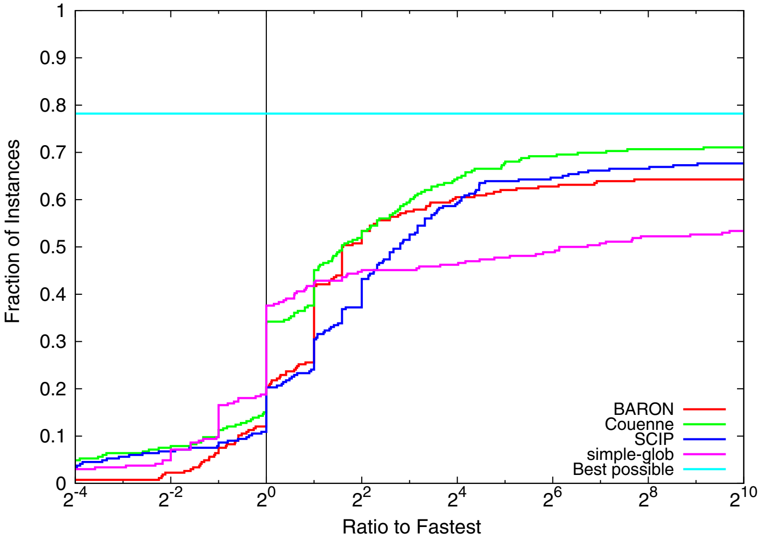

Fig. 11 Extended performance profile based on CPU time of global optimization solvers on 266 QCQP problems

Figure 11 plots the extended performance profiles. Our `simple-glob` method was able to solve over 50% of the problems in the time limit without using any heuristics, convexity-detection routines, cutting planes, or advanced presolving. Other solvers that use these techniques were able to solve nearly 70% of the instances.

5 Extensions of Minotaur

In this section, we provide a small set of examples that illustrate how Minotaur can be used and extended to exploit problem specific structures. In each case, we present the main algorithmic idea, show how it is implemented in Minotaur, and give numerical results from our experiments. The experimental setup used for this section is same as that described in Sect. 4.4.

5.1 Nonlinear presolve

Our first example illustrates how the availability of a native computational graph (see Sect. 3) allows us to discover nonlinear structure and to perform certain nonlinear reformulations that are simple but have a dramatic effect on the solution of an important class of MINLP problems.

We consider a chemical engineering problem from the IBM/CMU collection [16], namely, `syn20M04M`. This problem has 160 integer variables, 260 continuous variables, 56 convex nonlinear constraints, and 996 linear constraints. The NLP relaxation can be solved in a fraction of a second. However, standard NLPBB and LP/NLP-based solvers fail to solve this MINLP problem in a two-hour time limit.

Analysis of these models reveals that these problems contain big-M constraints of the form

$$
\begin{aligned}
& c(x_0, x_1, x_2, \ldots, x_k) \leq M_0(1 - x_0), \\
& 0 \leq x_i \leq M_i x_0, \quad i = 1, \ldots, k, \\
& x_0 \in \{0, 1\}, \; x_i \in \mathbb{R}, \; i = 1, \ldots, k.
\end{aligned}
\tag{5.1}
$$

This structure is common in MINLP. The binary variable x_0 acts as a “switch”, such that, when $x_0 = 1$ the continuous variables $x_1, \ldots, x_k$ can take nonzero values and the nonlinear constraint is enforced, and when $x_0 = 0$ the continuous variables are forced to zero, and M_0 is large enough so that the nonlinear constraint is redundant. The difficulty for MINLP solvers arises because the upper bound M_0 is not always “tight” when the nonlinear constraint is switched off. In particular, if M_0 is chosen to be too large, the continuous relaxation allows more noninteger solutions, thus making the relaxation weak.

To compute a tighter upper bound M_0, we exploit the implication constraints $0 \leq x_i \leq M_i x_0, i = 1, \ldots, k$. If $x_0 = 0$, then each $x_i = 0$ and we can replace the coefficient M_0 by

$$
c^u = c(0, \ldots, 0)
$$

if $c^u \leq M_0$. If $c^u > M_0$, we can fix $x_0 = 1$ since (5.1) is infeasible if $x_0 = 0$. The bound c^u obtained by exploiting the above structure is the tightest possible. The effect of improving the coefficient is dramatic. As shown in Table 6, the solution time for `syn20M04M` is reduced to about 66 seconds. To compute the tightest upper bound in

Table 6 Effects of presolve on the size of instance `syn20M04M` and its solution time

	No presolve	Basic presolve	Full presolve
Variables	420	328	292
Binary vars.	160	144	144
Constraints	1052	718	610
Nonlin. constr.	56	56	56
Bonmin-bnb (s)	$>$ 3600	NA	NA
Bonmin-QG (s)	$>$ 3600	NA	NA
Simple-bnb (s)	$>$ 3600	>3600	66.76
Simple-qg (s)	$>$ 3600	>3600	0.49

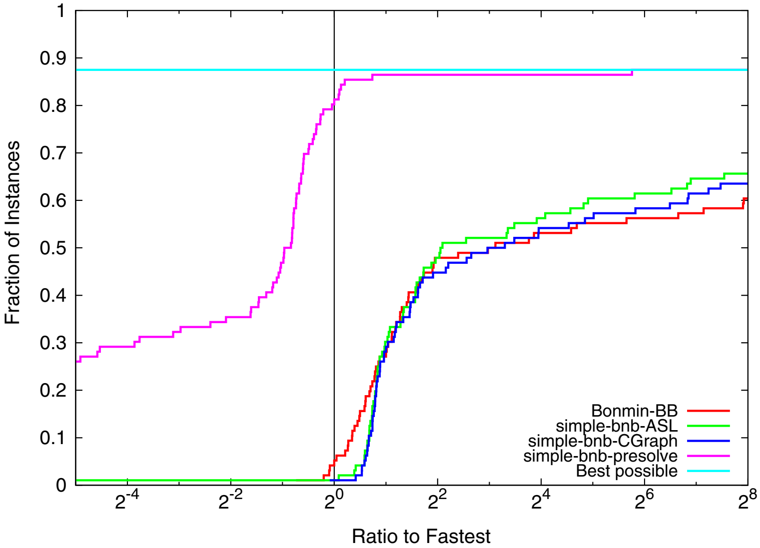

Fig. 12 Performance profile comparing presolve on `RSyn-X` and `Syn-X` instances

the general case, we need to solve the following nonconvex optimization problem:

$$\underset{x}{\text{maximize}} \;\; c(x_0, x_1, x_2, \ldots, x_k) \quad \text{subject to} \;\; x_0 = 0, x \in X,$$

where X is a feasible region of the MINLP problem obtained by removing constraint (5.1). This maximization problem can be efficiently solved in some cases (e.g., by exploiting the structure of the constraints).

We also compared the effect of this simple presolve technique on all 96 instances of `RSyn-*` and `Syn-*`. The results are shown in the performance profile in Fig. 12. This presolve clearly has a dramatic effect on the solution times for almost all of these instances, increasing the robustness by nearly 20% compared with that of Bonmin and our `simple-bnb` without presolve.

```
SolveStatus Presolver::solve()
{
  iters=0; subiters=0; last_ch_subiter = 0;
  while (iters<iterLimit_){
    ++iters;
    for (HandlerIterator h=handlers_.begin();h!=handlers_.end();++h){
      ++subiters; changed = false;
      h_status = (*h)->presolve(&mods_, &changed);
      if (h_status==SolvedOptimal || h_status==SolvedInfeasible ||
          h_status==SolvedUnbounded) {
        return h_status;
      }
      if (changed == true) last_ch_subiter = subiters;
      else if (subiters>=last_ch_subiter + handlers_.size()){
        return Finished
      }
    }
  }
  return Finished;
}
```

Fig. 13 Code snippet illustrating `Presolver:solve` function from base classes

We have implemented these and other more standard presolve techniques, well known from MILP [68], into the core Minotaur library. The `Presolver` class implements the main routine of calling various handlers for exploiting specific structures. In its simplest form, it calls the `Handler::presolve()` function of each of the handlers one by one repeatedly. It stops when no changes are made in the last k calls to handlers, where k is the number of handlers, or when a certain overall iteration limit is reached (see Fig. 13 for pseudocode). The handlers used in presolving may be specifically designed for presolving alone and need not implement every function in the `Handler` class (e.g., checking feasibility of a given point or separating a given point). For example, `NlPresHandler` class does not implement any other `Handler` functions besides presolving techniques for general nonlinear constraints and the objective function.

`NlPresHandler` applies four presolve methods to the problem: bound-improvement, identifying redundant constraints, coefficient improvement, and linearization of bilinear expressions on binary variables. In bound improvement, we consider all nonlinear constraints of the type $l_i \leq c_i(x) \leq u_i$. Using available bounds on the variables x, we find lower and upper bounds (L_i, U_i) of the nonlinear function by propagating the bounds in the forward-mode transversal of the computational graph. We then traverse in reverse mode to update bounds on each node of the graph based on the constraint bounds l_i and u_i. The bounds on variables are thus tightened. Additionally, if $l_i \leq L_i$ and $u_i \geq U_i$, then the constraint is identified as redundant and may be dropped. Similarly, if $L_i > u_i$ or $U_i < l_i$ for some i, then no point can satisfy this constraint, and the problem is infeasible.

When we are unable to find implications to fix all variables of the function c in (5.1), we find upper bounds on c^u by traversing the computational graph of c as in the bound-improvement technique. If we have a quadratic function of the form $\sum_{i,j} q_{ij} x_i x_j$ in which all variables have finite bounds and every term of the sum contains at least one binary variable, then `NlPresHandler` replaces this expression by an equivalent set

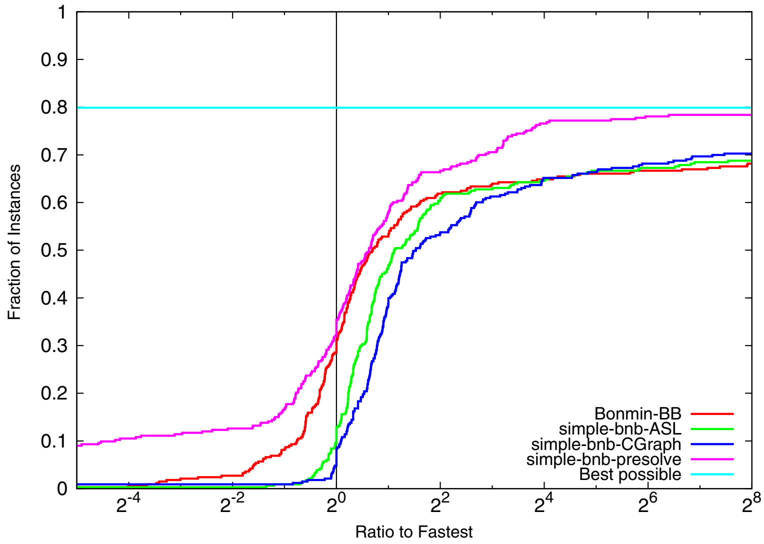

Fig. 14 Performance profile showing the impact of presolving on NLPBB solvers

of linear constraints. For example, the constraint $y_{1,2} = x_1x_2$, where $x_1 \in \{0, 1\}$ and $x_2 \in [l_2, u_2]$, is reformulated as

$$\begin{aligned} y_{1,2} &\le u_2x_1, \\ y_{1,2} &\ge l_2x_1, \\ x_2 - y_{1,2} &\le u_2(1 - x_1), \\ x_2 - y_{1,2} &\ge l_2(1 - x_1). \end{aligned}$$

Coefficient improvement, bound tightening, and identification of redundancy are also applied to linear constraints by the `LinearHandler`. In addition to these functions, the `LinearHandler` class implements dual fixing and identification of duplicate columns (variables) and constraints.

Figure 14 shows the effect of presolving on `simple-bnb` when applied to the 333 convex MINLP instances from MINLPLib-2. In this example, we call the presolver routines only once before the root node is solved in the branch-and-bound portion. Our `simple-bnb` solver with initial presolving was able to solve approximately 10% more instances than did the solver without presolve in the one-hour time limit. Similarly our `simple-qg` solver was able to solve 5% more instances with initial presolving (see Fig. 15).

5.2 Nonlinear perspective formulations

Consider the mixed binary set $\mathcal{S} = \{(x, z) \in \mathbb{R}^n \times \{0, 1\} : c(x) \le 0, lz \le x \le uz\}$, where c is a convex function. Setting the single binary variable z to zero forces all the other continuous variables x to zero. Moreover, the continous relaxation of $\mathcal{S}$ is a

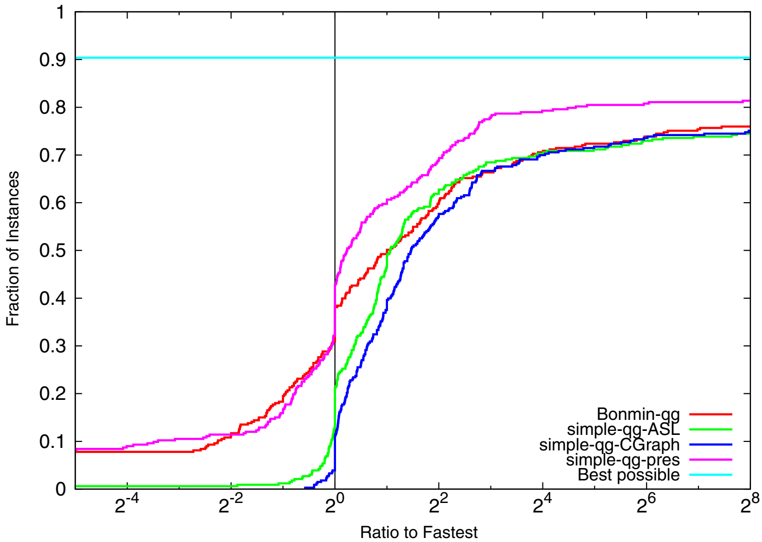

Fig. 15 Performance profile showing the impact of presolving on LP/NLP-based solvers

convex set. The convex hull of $\mathcal{S}$ can be obtained by taking a perspective reformulation of the nonlinear constraint [32]. More precisely, $conv(\mathcal{S}) = \{(x, z) \in \mathbb{R}^n \times [0, 1] : zc(\frac{x}{z}) \leq 0, lz \leq x \leq uz\}$. Perspective cuts using this reformulation can be derived and were shown to be effective in structured problem instances [32]. Perspective reformulations, cuts derived from these reformulations, and their applications have been studied extensively (see, e.g., [31,33,34,42–44]).

Replacing the original constraint $c(x) \leq 0$ with $zc(\frac{x}{z}) \leq 0$ creates difficulties for convex NLP solvers since the new function and its gradient need to be specially defined at $z = 0$ and constraint qualifications fail. The following approximation [35] overcomes these difficulties:

$$(z(1-\epsilon)+\epsilon)\, c\left(\frac{x}{z(1-\epsilon)+\epsilon}\right) \leq 0. \tag{5.2}$$

The function in (5.2) has the same value as that of c when $z = 0$ and $z = 1$. Moreover, this function is convex, and its gradient is well defined at $z = 0$. For small values of ϵ, this reformulation provides a good approximation to the convex hull while being amenable to solution by standard convex NLP solvers.

We implemented a function in the `NlPresHandler` class of Minotaur to identify this structure automatically. For each nonlinear constraint in the problem, Algorithm 3 is used to identify a binary variable for applying the reformulation. A set $\mathcal{C}$ of candidates is initially populated with all binary variables in the problems. For each variable x_i of this nonlinear constraint, we try to find those binary variables that turn x_i off by visiting all linear constraints that reference x_i. All other binary variables are removed from $\mathcal{C}$, and the procedure is repeated for the remaining variables in the nonlinear constraint. If a required binary variable is found, the original function is replaced by its approximate

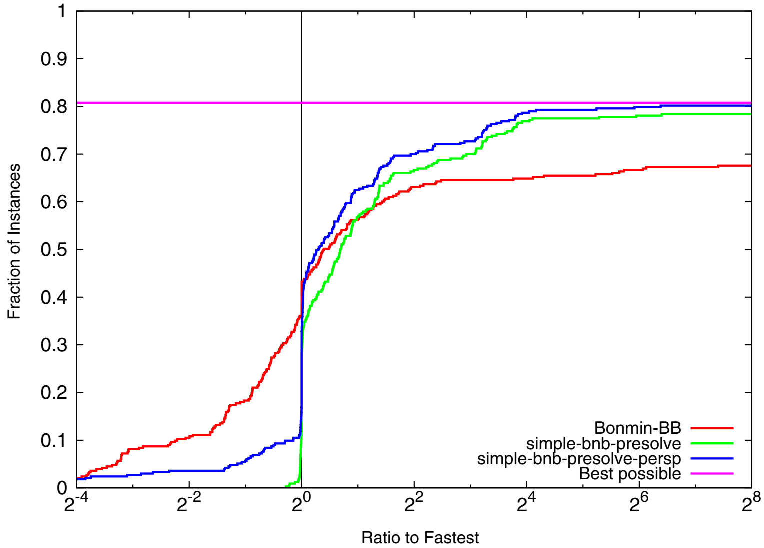

Fig. 16 Performance profile showing the impact of using perspective reformulations with NLPBB algorithms

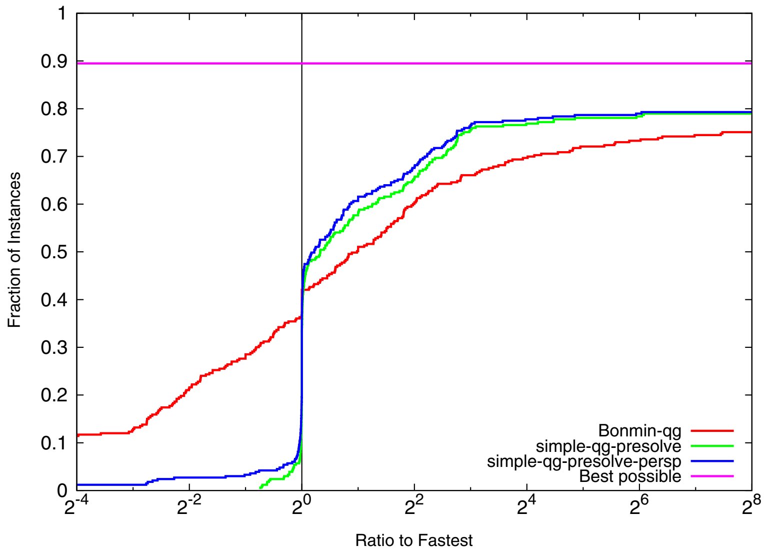

Fig. 17 Performance profile showing the impact of using perspective reformulations with LP/NLP-based methods

perspective reformulation (5.2) by updating the `CGraph` using a procedure similar to that in Fig. 4. This routine is applied only during the initial presolve phase.

Figures 16 and 17 show the effect of using the perspective reformulation with `simple-bnb` and `simple-qg`, respectively, on the convex problems in the MINLPLib-2 collection. Six instances that were not solved in the one-hour time limit with presolve using the `simple-bnb` code were solved after perspective reformulation. As one might expect from a tighter formulation, there was generally a reduction in the number of nodes. However, the reduction in time to solve the problems was less than the reduction in the number of nodes, as the average time spent for solving the NLP relaxations generally increased. Table 7 reports the time spent per NLP solve, the total number of nodes, and the total time to solve such instances. The reported time spent per NLP problem is not equal to the total time divided by the number of nodes because more than one NLP problem is solved at some nodes to find branching candidates.

Algorithm: `Detecting Perspective Structure`

input : A MINLP (1.1) and an index r of a nonlinear constraint in (1.1).
output: A binary variable that can be used for perspective reformulation of $c_r(x) \le 0$
begin
  $\mathcal{C} \longleftarrow \{t : x_t$ is a binary variable in (1.1)$\}$
  for *each i such that x_i appears in constraint $c_r(x) \le 0$* **do**
    $\mathcal{F}^i \longleftarrow \phi$ (Set of binary variables that turn off x_i)
    for *each j such that x_i appears in constraint $c_j(x) \le 0$ and c_j is a linear function* **do**
      Let $\mathcal{K}$ be the set of all binary variables in $\mathcal{C}$ that also appear in c_j
      for *each k in $\mathcal{K}$* **do**
        if *Fixing x_k to zero in c_j forces x_i to zero* **then**
          $\mathcal{F}^i \longleftarrow \mathcal{F}^i \cup \{k\}$
    $\mathcal{C} \longleftarrow \mathcal{C} \cap \mathcal{F}^i$
    if $\mathcal{C} = \phi$ **then**
      return *no variable found*
  return *first element of $\mathcal{C}$*

Algorithm 3: Algorithm for finding a binary variable that can be used to perform a perspective reformulation of a given nonlinear constraint.

6 Conclusions

Identifying and exploiting structure in MINLP problems are essential when solving difficult problems. A flexible and extensible framework for developing MINLP solvers is necessary in order to rapidly adopt new ideas and techniques and to specialize the methods. The modular class structure of our Minotaur framework provides such capabilities to developers and is the vehicle by which we deliver the resulting numerical methods to users. This flexibility does not come at the cost of speed and efficiency. As demonstrated by our nonlinear presolve and perspective formulation extensions, exploiting the problem structure in Minotaur can result in more reliable and efficient solvers. The source code for Minotaur is available from https://github.com/minotaur-solver/minotaur. Because of its availability and extensibility, Minotaur opens the door for future research to further advance the state of the art in algorithms and software for MINLP.

Table 7 Effect of using perspective reformulation on three measures: the total time taken (s) to solve an instance, the number of nodes processed in branch and bound, and the average time taken per NLP problem (ms)

Instance	Without perspective Ref.			With perspective Ref.		
	Time (s)	Nodes	Time/NLP (ms)	Time (s)	Nodes	Time/NLP (ms)
rsyn0805m	33.38	1103	24.35	30.59	802	29.96
rsyn0805m02m	873.88	7444	103.20	774.96	5387	121.67
rsyn0805m03m	2259.38	12,303	163.44	1943.15	8485	195.62
rsyn0805m04m	>3600.00	13,832	227.78	>3600.00	8814	332.13
rsyn0810m	26.96	786	26.53	31.44	680	36.07
rsyn0810m02m	1030.63	8404	108.99	1014.40	5874	146.51
rsyn0810m03m	>3600.00	16,642	194.67	1879.59	6257	239.25
rsyn0810m04m	>3600.00	10,875	273.70	2049.40	3741	368.59
rsyn0815m	23.70	505	30.09	29.38	495	41.32
rsyn0815m02m	1259.49	8617	128.12	1342.10	6294	185.64
rsyn0815m03m	>3600.00	15,640	206.27	2783.14	8113	289.00
rsyn0815m04m	>3600.00	9568	296.64	>3600.00	6407	428.55
rsyn0820m	35.34	822	30.35	25.02	406	42.52
rsyn0820m02m	1742.62	10,084	152.14	1190.27	5027	194.82
rsyn0820m03m	>3600.00	13,476	230.98	>3600.00	9099	335.58
rsyn0820m04m	>3600.00	7378	355.32	>3600.00	5380	489.58
rsyn0830m	30.86	576	35.67	27.46	301	55.55
rsyn0830m02m	3214.76	17,294	171.81	1007.66	3358	225.94
rsyn0830m03m	>3600.00	10,329	289.15	2993.20	5492	423.63
rsyn0830m04m	>3600.00	5420	439.46	>3600.00	4063	609.82
rsyn0840m	71.12	1292	44.93	43.04	452	67.40
rsyn0840m02m	>3600.00	16,059	202.69	1234.59	3670	266.33
rsyn0840m03m	>3600.00	8205	336.70	1404.32	1924	415.15
rsyn0840m04m	>3600.00	4148	485.52	>3600.00	3438	680.76
syn05m	0.07	5	7.78	0.06	3	12.00
syn05m02m	0.32	7	15.24	0.20	3	22.22
syn05m03m	0.75	11	22.73	0.41	5	31.54
syn05m04m	1.25	15	30.49	0.68	7	40.00
syn10m	0.06	3	12.00	0.09	3	18.00
syn10m02m	1.32	13	24.91	0.58	5	34.12
syn10m03m	3.48	21	40.94	1.33	9	53.20
syn10m04m	6.11	27	57.01	2.39	13	72.42
syn15m	0.12	5	13.33	0.12	3	24.00
syn15m02m	0.60	5	35.29	0.47	3	52.22
syn15m03m	1.64	9	56.55	0.99	5	76.15
syn15m04m	2.97	13	80.27	1.87	7	110.00

Table 7 continued

Instance	Without perspective Ref.			With perspective Ref.		
	Time (s)	Nodes	Time/NLP (ms)	Time (s)	Nodes	Time/NLP (ms)
syn20m	0.50	15	15.15	0.26	5	23.64
syn20m02m	2.60	15	43.90	1.87	7	60.32
syn20m03m	20.48	79	72.85	4.06	13	94.42
syn20m04m	66.76	150	109.38	7.33	19	133.27
syn30m	1.11	27	19.47	0.54	7	31.76
syn30m02m	6.88	23	64.30	3.39	9	82.68
syn30m03m	57.29	148	100.49	7.93	11	130.00
syn30m04m	210.63	389	146.65	16.06	17	188.94
syn40m	2.00	43	25.97	1.04	11	41.60
syn40m02m	23.04	80	81.13	9.00	11	103.45
syn40m03m	272.23	859	135.93	18.70	34	179.81
syn40m04m	318.25	460	194.50	49.74	63	260.42

Acknowledgements This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract DE-AC02-06CH11357. This work was also supported by the U.S. Department of Energy under Grant DE-FG02-05ER25694.

References

1. Abhishek, K., Leyffer, S., Linderoth, J.T.: FilMINT: an outer-approximation-based solver for nonlinear mixed integer programs. INFORMS J. Comput. **22**, 555–567 (2010). https://doi.org/10.1287/ijoc.1090.0373
2. Achterberg, T.: SCIP: solving constraint integer programs. Math. Program. Comput. **1**(1), 1–41 (2009)
3. Achterberg, T., Koch, T., Martin, A.: Branching rules revisited. Oper. Res. Lett. **33**, 42–54 (2004)
4. Adjiman, C.S., Androulakis, I.P., Floudas, C.A.: A global optimization method, αBB, for general twice-differentiable constrained NLPs-II. Implementation and computational results. Comput. Chem. Eng. **22**, 1159–1179 (1998)
5. Beale, E.W.L., Tomlin, J.A.: Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. In: Lawrence, J. (ed.) Proceedings of the 5th International Conference on Operations Research, pp. 447–454 (1969)
6. Belotti, P.: COUENNE: A user's manual. Technical report. Lehigh University (2009)
7. Belotti, P., Kirches, C., Leyffer, S., Linderoth, J.T., Luedtke, J., Mahajan, A.: Mixed-integer nonlinear optimization. Acta Numer. **22**, 1–131 (2013)
8. Benson, H.Y.: Mixed integer nonlinear programming using interior point methods. Optim. Methods Softw. **26**(6), 911–931 (2011)
9. Bonami, P., Biegler, L.T., Conn, A.R., Cornuéjols, G., Grossmann, I.E., Laird, C.D., Lee, J., Lodi, A., Margot, F., Sawaya, N., Wächter, A.: An algorithmic framework for convex mixed integer nonlinear programs. Discrete Optim. **5**(2), 186–204 (2008)
10. Bonami, P., Lee, J., Leyffer, S., Wächter, A.: On branching rules for convex mixed-integer nonlinear optimization. J. Exp. Algorithm. **18**, 2.6:2.1–2.6:2.31 (2013)
11. Brooke, A., Kendrick, D., Meeraus, A., Raman, R.: GAMS, A User's Guide. GAMS Development Corporation, Fairfax (1992)
12. Bussieck, M.R., Drud, A.: SBB: a new solver for mixed integer nonlinear programming. Talk, OR 2001, Section Continuous Optimization (2001)

13. Bussieck, M.R., Drud, A., Meeraus, A.: MINLPLib—a collection of test models for mixed-integer nonlinear programming. INFORMS J. Comput. **15**(1), 114–119 (2003)
14. Byrd, R.H., Nocedal, J., Richard, W.A.: KNITRO: an integrated package for nonlinear optimization. In: Pillo, G., Roma, M. (eds.) Large-Scale Nonlinear Optimization, Volume 83 of Nonconvex Optimization and Its Applications, pp. 35–59. Springer, Berlin (2006)
15. Christianson, B.: Automatic Hessians by reverse accumulations. IMA J. Numer. Anal. **12**(2), 135–150 (1992)
16. CMU-IBM cyber-infrastructure for MINLP (2009). http://www.minlp.org/
17. COIN-OR: Computational Infrastructure for Operations Research (2014). http://www.coin-or.org
18. Dakin, R.J.: A tree search algorithm for mixed programming problems. Comput. J. **8**, 250–255 (1965)
19. Dolan, E., Moré, J.: Benchmarking optimization software with performance profiles. Math. Program. **91**, 201–213 (2002)
20. Drewes, S.: Mixed Integer Second Order Cone Programming. Ph.D. thesis. Technische Universität Darmstadt (2009)
21. Drewes, S., Ulbrich, S.: Subgradient based outer approximation for mixed integer second order cone programming. Mixed Integer Nonlinear Programming, Volume 154 of the IMA Volumes in Mathematics and Its Applications, pp. 41–59. Springer, New York (2012)
22. Duran, M.A., Grossmann, I.: An outer-approximation algorithm for a class of mixed-integer nonlinear programs. Math. Program. **36**, 307–339 (1986)
23. Ferreau, H.J., Kirches, C., Potschka, A., Bock, H.G., Diehl, M.: qpOASES: a parametric active-set algorithm for quadratic programming. Math. Program. Comput. **6**(4), 327–363 (2014)
24. Fletcher, R.: User Manual for BQPD. University of Dundee, Dundee (1995)
25. Fletcher, R., Leyffer, S.: Solving mixed integer nonlinear programs by outer approximation. Math. Program. **66**, 327–349 (1994)
26. Fletcher, R., Leyffer, S.: User Manual for filterSQP. University of Dundee Numerical Analysis Report NA-181 (1998)
27. Fletcher, R., Leyffer, S.: Nonlinear programming without a penalty function. Math. Program. **91**, 239–270 (2002)
28. Floudas, C.A.: Deterministic Global Optimization: Theory, Algorithms and Applications. Kluwer Academic Publishers, Dordrecht (2000)
29. Forrest, J.: CLP (2014). http://www.coin-or.org/
30. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: A Modeling Language for Mathematical Programming. The Scientific Press, Cambridge (1993)
31. Frangioni, A., Furini, F., Gentile, C.: Approximated perspective relaxations: a project and lift approach. Comput. Optim. Appl. **63**, 705–735 (2016). https://doi.org/10.1007/s10589-015-9787-8
32. Frangioni, A., Gentile, C.: Perspective cuts for a class of convex 0–1 mixed integer programs. Math. Program. **106**, 225–236 (2006)
33. Frangioni, A., Gentile, C.: SDP diagonalizations and perspective cuts for a class of nonseparable MIQP. Oper. Res. Lett. **35**, 181–185 (2007)
34. Frangioni, A., Gentile, C.: A computational comparison of reformulations of the perspective relaxation: SOCP vs cutting planes. Oper. Res. Lett. **37**(3), 206–210 (2009)
35. Furman, K., Grossmann, I., Sawaya, N.: An exact MINLP formulation for nonlinear disjunctive programs based on the convex hull. In: Presented at the 20th International Symposium on Mathematical Programming, Chicago, IL (2009)
36. Gay, D.M.: More AD of nonlinear AMPL models: computing Hessian information and exploiting partial separability. In: Berz, M., Bischof, C., Corliss, G., Griewank, A. (eds.) Computational Differentiation Techniques Applications and Tools. SIAM, Philadelphia (1996)
37. Gebremedhin, A.H., Tarafdar, A., Pothen, A., Walther, A.: Efficient computation of sparse Hessians using coloring and automatic differentiation. INFORMS J. Comput. **21**(2), 209–223 (2009)
38. Geoffrion, A.M.: Generalized Benders decomposition. J. Optim. Theory Appl. **10**(4), 237–260 (1972)
39. Gould, N., Scott, J.: A note on performance profiles for benchmarking software. ACM Trans. Math. Softw. **43**(2), 1–5 (2016)
40. Griewank, A., Walther, A.: Evaluating Derivatives Principles and Techniques of Algorithmic Differentiation, Second edn. SIAM, Philadelphia (2008)
41. 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. Springer, New York (1997)

42. Günlük, O., Linderoth, J.: Perspective relaxation of mixed integer nonlinear programs with indicator variables. In: Lodi, A., Panconesi, A., Rinaldi, G. (eds.) IPCO 2008: The Thirteenth Conference on Integer Programming and Combinatorial Optimization, vol. 5035, pp. 1–16 (2008)
43. Günlük, O., Linderoth, J.T.: Perspective relaxation of mixed integer nonlinear programs with indicator variables. Math. Program. Ser. B **104**, 186–203 (2010)
44. Günlük, O., Linderoth, J.T.: Perspective reformulation and applications. IMA Vol. **154**, 61–92 (2012)
45. Gupta, O.K., Ravindran, A.: Branch and bound experiments in convex nonlinear integer programming. Manag. Sci. **31**, 1533–1546 (1985)
46. Gurobi Optimization, Inc. Gurobi Optimizer Reference Manual, Version 5.6 (2014)
47. Hart, W.E., Watson, J.-P., Woodruff, D.L.: Pyomo: modeling and solving mathematical programs in Python. Math. Program. Comput. **3**, 219–260 (2011)
48. IBM Corp. IBM ILOG CPLEX V12.6: User's Manual for CPLEX (2014)
49. Jeroslow, R.G.: There cannot be any algorithm for integer programming with quadratic constraints. Oper. Res. **21**(1), 221–224 (1973)
50. Kannan, R., Monma, C.L.: On the computational complexity of integer programming problems. In: Henn, R., Korte, B., Oettli, W. (eds.) Optimization and Operations Research, Volume 157 of Lecture Notes in Economics and Mathematical Systems, pp. 161–172. Springer, Berlin (1978)
51. Land, A.H., Doig, A.G.: An automatic method for solving discrete programming problems. Econometrica **28**, 497–520 (1960)
52. Lasserre, J.: An explicit exact SDP relaxation for nonlinear 0–1 programs. In: Aardal, K., Gerards, A.M.H. (eds.) Integer Programming and Combinatorial Optimization 2001, Lecture Notes in Computer Science, vol. 2081, pp. 293–303. Springer, Berlin (2001)
53. Lasserre, J.: Global optimization with polynomials and the problem of moments. SIAM J. Optim. **11**(3), 796–817 (2001)
54. Laurent, M.: A comparison of the Sherali–Adams, Lovász–Schrijver, and Lasserre relaxations for 0–1 programming. Math. Oper. Res. **28**(3), 470–496 (2003)
55. Leyffer, S.: User Manual for MINLP-BB. University of Dundee, Dundee (1998)
56. Leyffer, S.: Mixed-Integer PDE-Constrained Optimization. Technical report. Argonne (2015)
57. Leyffer, S., Linderoth, J.T., Luedtke, J., Miller, A., Munson T.: Applications and algorithms for mixed integer nonlinear programming. In: Journal of Physics: Conference Series, SciDAC 2009, vol. 180, pp. 012014 (2009)
58. Mahajan, A., Leyffer, S., Kirches, C.: Solving convex mixed-integer nonlinear programs by QP-diving. Preprint ANL/MCS-P1801-101. Argonne National Laboratory (2010)
59. Mahajan, A., Leyffer, S., Kirches, C.: Solving mixed-integer nonlinear programs by QP-diving. Preprint ANL/MCS-2071-0312. Argonne National Laboratory, Mathematics and Computer Science Division (2012)
60. McCormick, G.P.: Computability of global solutions to factorable nonconvex programs: part I—convex underestimating problems. Math. Program. **10**, 147–175 (1976)
61. Misener, R., Floudas, C.A.: GloMIQO: global mixed-integer quadratic optimizer. J. Glob. Optim. **5**, 1–48 (2012)
62. Misener, R., Floudas, C.A.: ANTIGONE: algorithms for coNTinuous/integer global optimization of nonlinear equations. J. Glob. Optim. **59**, 503–526 (2014)
63. Neumaier, A.: Complete search in continuous global optimization and constraint satisfaction. Acta Numer. **13**, 271–369 (2004)
64. Nocedal, J., Wright, S.J.: Numerical Optimization. Springer, New York (1999)
65. Quesada, I., Grossmann, I.E.: An LP/NLP based branch-and-bound algorithm for convex MINLP optimization problems. Comput. Chem. Eng. **16**, 937–947 (1992)
66. Ryoo, H.S., Sahinidis, N.V.: Global optimization of nonconvex NLPs and MINLPs with applications in process design. Comput. Chem. Eng. **19**, 552–566 (1995)
67. Sahinidis, N.V.: BARON: a general purpose global optimization software package. J. Glob. Optim. **8**, 201–205 (1996)
68. Savelsbergh, M.W.P.: Preprocessing and probing techniques for mixed integer programming problems. ORSA J. Comput. **6**, 445–454 (1994)
69. Schichl, H.: Global optimization in the COCONUT project. In: Alt, R., Frommer, A., Baker Kearfott, R., Luther, W. (eds.) Numerical Software with Result Verification, Volume 2991 of Lecture Notes in Computer Science, pp. 243–249. Springer, Berlin (2004)

70. Still, C., Westerlund, T.: Solving convex MINLP optimization problems using a sequential cutting plane algorithm. Comput. Optim. Appl. **34**(1), 63–83 (2006)
71. Tawarmalani, M., Sahinidis, N.V.: Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming: Theory, Algorithms, Software, and Applications. Kluwer Academic Publishers, Boston (2002)
72. Van Roy, T.J.: Cross decomposition for mixed integer programming. Math. Program. **25**, 145–163 (1983)
73. Vigerske, S.: MINLPLib 2. In: Proceedings of the XII Global Optimization Workshop: Mathematical and Applied Global Optimization, pp. 137–140 (2014)
74. 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 (2018). https://doi.org/10.1080/10556788.2017.1335312
75. Viswanathan, J., Grossmann, I.E.: A combined penalty function and outer-approximation method for MINLP optimization. Comput. Chem. Eng. **14**(7), 769–782 (1990)
76. 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)
77. Westerlund, T., Lundqvist, K.: Alpha-ECP, version 5.01: an interactive MINLP-solver based on the extended cutting plane method. Technical Report 01-178-A. Process Design Laboratory at Åbo University (2001)
78. Westerlund, T., Pettersson, F.: A cutting plane method for solving convex MINLP problems. Comput. Chem. Eng. **19**, 131–136 (1995)
79. Ziena Optimization. KNITRO Documentation (2012)