Computational Optimization and Applications (2022) 81:423–478
https://doi.org/10.1007/s10589-021-00335-x

Linearization and parallelization schemes for convex mixed-integer nonlinear optimization

Meenarli Sharma[1] · **Prashant Palkar**[1] · **Ashutosh Mahajan**[1]

Received: 14 May 2020 / Accepted: 14 November 2021 / Published online: 20 January 2022

Abstract

We develop and test linearization and parallelization schemes for convex mixed-integer nonlinear programming. Several linearization approaches are proposed for LP/NLP based branch-and-bound. Some of these approaches strengthen the linear approximation to nonlinear constraints at the root node and some at the other branch-and-bound nodes. Two of the techniques are specifically applicable to commonly found univariate nonlinear functions and are more effective than other general approaches. These techniques have been implemented in the Minotaur toolkit. Tests on benchmark instances show up to 12% improvement in the average time to solve the instances. Shared-memory parallel versions of NLP based branch-and-bound and LP/NLP based branch-and-bound algorithms have also been developed in the toolkit. These implementations solve different nodes of branch-and-bound concurrently. About 44% improvement in the speed and an increase in the number of instances solved within the time limit are observed when the two schemes are used together on a computer with 16 cores. These parallelization methods are compared to alternate approaches that exploit parallelism in existing commercial MILP solvers. The latter approaches are seen to perform better thus highlighting the importance of MILP techniques.

Keywords Convex MINLP · Linearization techniques · Branch-and-bound · Outer approximation · Shared-memory parallel

✉ Meenarli Sharma
meenarli@iitb.ac.in

Prashant Palkar
prashant.palkar@iitb.ac.in

Ashutosh Mahajan
amahajan@iitb.ac.in

[1] Indian Institute of Technology Bombay, Mumbai, MH 400076, India

1 Introduction

Mixed-Integer Nonlinear Programs (MINLPs) are optimization problems with nonlinear objective or constraint functions and some integer constrained variables. While MINLP applications arise in several domains, they are difficult to solve. Special cases of MINLP, like Mixed-Integer Linear Programs (MILPs) and nonconvex global optimization are themselves NP-hard in general. We refer the interested readers to recent surveys [9, 15, 31, 48]) for an overview of applications, solution methods and computational complexity of MINLPs. We consider the special case of convex MINLPs only, which is still hard, but is more tractable than the general case. In particular, we consider the following problem:

$$\begin{aligned} \min_{x} \; & f(x) \\ \text{s.t. } & g(x) \leq b, \\ & x \in X, \\ & x_j \in \mathbb{Z}, \quad \forall j \in \mathcal{I}, \end{aligned} \tag{P}$$

where the given functions $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^m$ are convex, twice continuously differentiable on the polyhedral set $X := \{x : Cx \leq c, Dx = d\}$ where C, D, c and d are matrices and vectors of appropriate dimensions, and $\mathcal{I}$ is the index set of integer constrained decision variables.

Deterministic methods for convex MINLPs are based on branch-and-bound type of algorithms, just like those for MILPs. A branch-and-bound method starts by solving a relaxation of (P), that has a larger feasible region enclosing (P), but is easier to solve to global optimality. A solution of this relaxation provides a valid lower bound on the optimal value (say z^*) of (P). Then one divides the search-space by branching to create smaller subproblems. A relaxation of each subproblem is then solved. Each subproblem, a smaller relaxation than its parent, has a lower bound no less than its parent. If a solution to any of the subproblems is feasible for (P), its objective value provides an upper bound on z^*. The algorithm stops when the lower bound and the upper bound on z^* converge. This setup is easily viewed and analyzed as a tree-search where the tree-nodes denote the subproblems and the edges denote the branches that further divide a subproblem.

We focus on two approaches for enhancing the performance of algorithms for convex MINLPs: (a) creating better relaxations through effective linearization inequalities, and (b) using shared-memory parallel search to explore the branch-and-bound tree using multiple processors of a computer. Creating good relaxations, that provide a lower bound closer to z^*, in a reasonable amount of time is important for fast convergence of the branch-and-bound based algorithms. Rather than starting with a tight relaxation which may be difficult to solve, one can first solve a weaker relaxation and then tighten it iteratively by adding valid inequalities. Combining this scheme with branch-and-bound leads to what is called a branch-and-cut method which most solvers deploy for solving MILPs and MINLPs. A commonly used technique for creating linear relaxations of convex nonlinear constraints is through a gradient based linearization. Given a convex

differentiable nonlinear function $\hat{g} : \mathbb{R}^n \to \mathbb{R}$ and a point $x' \in \mathbb{R}^n$, the following well known gradient inequality [46]

$$\nabla \hat{g}(x')^T(x - x') + \hat{g}(x') \leq \hat{g}(x)$$

holds for all $x \in \mathbb{R}^n$. One can thus create a relaxation of (P) by replacing its nonlinear constraints by

$$\nabla g(x')^T(x - x') + g(x') \leq b. \tag{grad-I}$$

This relaxation can be tightened by adding linearization inequalities obtained from multiple points. We propose schemes that try to identify more effective linearization inequalities by finding suitable points of linearization.

Another way of speeding up algorithms for solving MINLPs is to exploit the availability of multiple processors that is common on modern computing architectures. We describe in the later half of this paper, a shared-memory parallel implementation of three algorithms for convex MINLPs: (i) NLP based branch-and-bound, (ii) two variants of LP/NLP based branch-and-bound and (iii) MILP based outer-approximation. We study the effects of different algorithmic components: sharing of information like branching scores amongst different threads, and scalability with the number of threads.

The above mentioned advancements have been implemented within the open-source Minotaur framework [39] and tested on benchmark instances from MINLPLib [16]. We next describe in Sect. 2 the algorithms and solvers for convex MINLPs. Section 3 describes the linearization schemes, and Sect. 4 presents the parallelization schemes. Computational results from combining the two approaches are presented in Sect. 5. Section 6 presents results of methods that deploy an MILP solver capable of using multiple CPUs for solving MILP relaxations. Section 8 contains our conclusions and scope for future work.

2 Algorithms and solvers for convex MINLPs

Methods for solving convex MINLPs primarily differ in the way they create a relaxation of the MINLP. We first describe the main algorithms and then briefly survey the solvers available.

2.1 Algorithms

Given a MINLP (P), the most natural option is to relax the integer restrictions on variables and obtain a convex nonlinear program (convex NLP):

$$\begin{aligned} \min_x \ & f(x) \\ \text{s.t. } & g(x) \leq b, \\ & x \in X. \end{aligned} \tag{R}$$

If relaxation (R) of (P) is infeasible, then so is (P). If the solution, say x^0, of the NLP relaxation satisfies integer restrictions of (P), then it is an optimal solution to

(P) as well. If, on the other hand, x^0 does not satisfy the integrality restrictions, we get a lower bound on the optimal value of (P). The nonlinear branch-and-bound [27] (NLP-BB) method proceeds by dividing the search-space into two or more subproblems in a way that every solution of (P) lies in at least one of the subproblems while x^0 does not lie in any of them. Each subproblem thus created is a smaller MINLP, and this process is continued recursively.

The outer-approximation (OA) algorithm [22] solves an alternating sequence of MILPs and NLPs. It is initialized by solving (R). If the solution x^0 is not integer feasible, the nonlinear functions are replaced by linearization inequalities (grad-I) obtained at x^0, and the integer restrictions are re-introduced to obtain the following MILP relaxation. If the objective function is also nonlinear, the problem is reformulated by replacing the objective with an auxiliary variable, η, and adding the constraint $f(x) \leq \eta$. This new constraint is also replaced by its linearization inequality at x^0 in the MILP relaxation:

$$\begin{aligned} \min_{x,\eta}\ & \eta \\ \text{s.t.}\ & \nabla f(x^0)^T(x - x^0) + f(x^0) \leq \eta, \\ & \nabla g(x^0)^T(x - x^0) + g(x^0) \leq b, \\ & x \in X, \\ & x_j \in \mathbb{Z},\ \forall j \in \mathcal{I}. \end{aligned} \tag{RM}$$

The MILP relaxation (RM) is solved using an MILP solver. If the MILP is infeasible, then so is (P). If the MILP solution (say, $\hat{x}$) satisfies all nonlinear constraints, then it is optimal to (P). Otherwise, the MILP optimal value (say, $\hat{z}$) provides a lower bound on z^*. Next, a 'fixed' NLP of the following form is solved.

$$\begin{aligned} \min_{x}\ & f(x) \\ \text{s.t.}\ & g(x) \leq\ b, \\ & x \in X, \\ & x_j = \hat{x}_j,\ \forall j \in \mathcal{I}. \end{aligned} \tag{F-NLP}$$

We denote this NLP as F-NLP($\hat{x}$) to indicate that the integer variables are fixed to the values in $\hat{x}$. An optimal solution to F-NLP($\hat{x}$) provides an upper bound on z^*. The optimal solution is then used to generate more linearization constraints (grad-I) that are added to the MILP relaxation. The updated MILP is solved again and the process is repeated. The new inequalities ensure that all solutions of MILP with $x_j = \hat{x}_j, j \in \mathcal{I}$ have objective value no less than $\hat{z}$. If the 'fixed' NLP is infeasible, the point returned by the NLP solvers can still be used to generate valid underestimators and linear constraints [23]. These linearization inequalities forbid the integer combination $\hat{x}_j, j \in \mathcal{I}$ in the future MILP solutions. Another related algorithm, Generalized Benders Decomposition (GBD) algorithm [25], generates a single inequality at the NLP solution which is then added to the MILP. Both OA and GBD do not require any implementation of tree-search unlike the NLP based branch-and-bound. They naturally exploit the advances that have been made in the MILP technology

over the decades, including presolving [5, 38], cutting planes [12, 34], heuristic search [10, 14], conflict analysis [3, 58] and parallel search [11, 52, 53] etc.

The LP/NLP based branch-and-cut algorithm of Quesada and Grossmann [43], which we also refer to as QG tries to overcome the difficulty of solving similar MILPs repeatedly. It creates and maintains a single branch-and-cut tree. Like OA, it starts by solving the NLP relaxation (R), and creates a linear relaxation of (P) by relaxing integrality from (RM). It then initiates the single-tree by solving this root LP relaxation of (P), and proceeds like LP based branch-and-cut method. When a node in the search-tree yields an integer optimal solution ($\hat{x}$), F-NLP($\hat{x}$) is solved. If the NLP is feasible, its optimal solution provides an upper bound on z^*. Linearization inequalities obtained at the point returned by solving F-NLP($\hat{x}$), say $\check{x}$, are added to all the open-nodes of the tree to tighten the relaxations, and branch-and-cut is resumed. While the algorithm is known to take a finite number of steps, careful implementation and control are required for it to be practically useful. Convex MINLPs are known to be NP-hard, and this algorithm, like others, can take a long time to run. In the later sections, we demonstrate effectiveness of some practical ideas that enhance the performance of this algorithm.

2.2 Solvers

The above mentioned algorithms and their variants have been implemented in several convex MINLP solvers including AIMMS [30], BONMIN [13], FilMINT [2], Muriqui [40], and SHOT [37]. Global solvers like Antigone [41], BARON [47], Couenne [8], LINDO [35] and SCIP [4] can also be used to solve convex MINLPs. Global solvers implement heuristics to detect convexity automatically and resort to slower methods for nonconvex problems if they fail to detect it. All the stated solvers except SCIP rely on a separate MILP solver for implementing branch-and-cut and related routines. The open-source Minotaur toolkit [39] is used to implement the methods proposed in this paper. Minotaur includes two solvers for convex MINLPs: NLP-BB and QG against which we compare the effects of the proposed schemes. While, it implements its own branch-and-cut, it also has the ability to interface with MILP solvers to use their implementation of branch-and-cut. The latter is used to implement OA and a variant of QG.

Use of shared-memory parallel computing for MILPs has received attention recently, see for example [18, 45, 54]. Most open-source [24, 44] and proprietary MILP solvers [60–64] exploit multiple processors for branch-and-bound/cut framework. Some of the frameworks that exploit shared-memory parallelization are Ubiquity Generator (UG) [49, 50], ChiPPS [59] and PEBBL [28]. The UG framework has been used as a parallelization wrapper over many MILP base solvers [11, 42, 51–53]. It explicitly controls the base solver as a callable library by parallelizing the tree-search from outside. FiberSCIP (FSCIP) is the shared-memory parallel algorithm that uses SCIP underneath UG. The frameworks ChiPPS and PEBBL use a master-hub-worker and a hub-worker hierarchy, respectively. The MILP solver, CBC [24] implements a multithreaded scheme to parallelize its sequential solver. Nodes are assigned by a master thread to workers sequentially as some of the global

data is stored centrally. It also has a deterministic parallelization mode which distributes subtrees to workers instead of nodes. Proprietary software like CPLEX and GUROBI provide LP solvers that can be used as subroutines for solving MINLPs. They also provide MILP solvers that can run in a parallel mode. CPLEX LP and MILP solvers are extensively used in our experiments.

2.3 Experimental setup

All the computational experiments have been carried out on a system with two 64-bit Intel(R) Xeon(R) E5-2670 v2, 2.50GHz CPUs having 10 cores each and sharing 128GB RAM. Hyperthreading is disabled. Our schemes are available in the development version of Minotaur.[1] All codes are complied with GCC-4.9.2 compiler. OpenMP-4.0 support provided by GCC is used for compiling parallel algorithms. IPOPT-3.12 with MA27 linear-systems solver is used as the NLP solver. CPLEX-12.8 has been used as the LP solver. CPLEX-12.8 MILP solver is used in algorithms that require solving an MILP. There are 374 instances in MINLPLib [16] that are known to be convex. We excluded 40 instances that did not have any nonlinearity (in constraints and objective) or any integer variables after the presolving step in Minotaur. We used the remaining 334 instances and refer to them as the *TS* test set in our experiments. Description of these instances is presented in Appendix A. We have set a limit of one hour on the wall clock time in all our experiments and reported all the solution times in seconds.

3 Linearization schemes

Recall that the QG algorithm creates an MILP relaxation of the nonlinear feasible region which is solved by branch-and-cut. Adding linearizations only when we reach integer feasible points in branch-and-bound tree may lead to a weak relaxation, and adding many of these early on can slow down the speed. We propose two sets of schemes - one for tightening the initial LP relaxation at the root node and the other for adding new linearizations later in the branch-and-bound tree. Strategies for generating linearizations based on the change in the lower bound, depth of the nodes in the search-tree, etc., and using NLP techniques for selecting points for linearizations have previously been proposed in [1, 32] for use in the FilMINT solver.

We analyzed performance of default QG in Minotaur on 267 instances in test set *TS* which have at least one nonlinear constraint and observed that a large fraction of the nodes processed yield fractional optimal solutions (Fig. 1 (left)), many of which also violate a large fraction of nonlinear constraints (Fig. 1 (right)). These observations motivated us to add more linearizations at selected nodes.

[1] Available at http://github.com/minotaur-solver/minotaur.

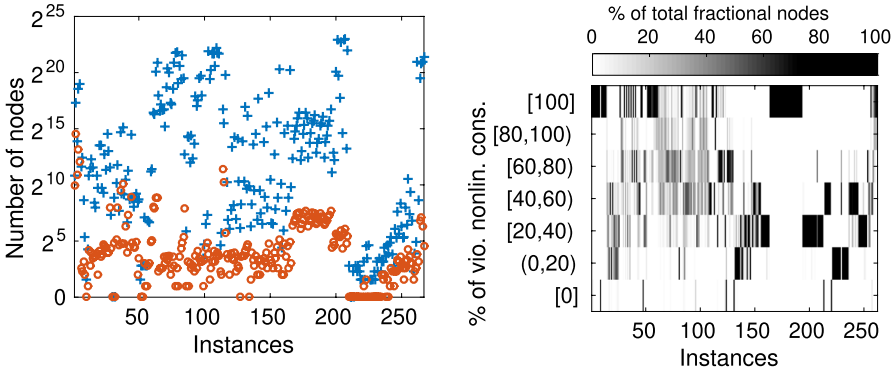

Fig. 1 (Left) Total number of nodes processed (+) and the number of nodes with integer LP optimal solution (o). (Right) Distribution of the violated nonlinear constraints at the nodes with fractional LP solution

3.1 Linearization techniques at the root node

Given a problem (P) and the solution x^0 of its continuous relaxation (R), let $\bar{P}_k$ be a polyhedron corresponding to the k^{th} nonlinear constraint $(g_k(x) \leq b_k)$ defined as,

$$\bar{P}_k := \{x : \nabla g_k(x^0)^T(x - x^0) + g_k(x^0) \leq b_k\}. \tag{1}$$

The feasible region of the root LP relaxation can be interpreted as an intersection of polyhedra $\bar{P}_k, k \in 1, \ldots, m$, corresponding to the nonlinear constraints, and X. In this section, we propose five schemes that aim to tighten the LP relaxation at the root node by tightening $\bar{P}_k, k = 1, \ldots, m$.

The first two schemes are designed for problems in which a constraint $g_k(x) \leq b_k$ has a univariate nonlinear structure, i.e., g_k is the sum of a univariate nonlinear function and a linear function, and the variable in the linear part of g_k do not appear in its nonlinear part. Mathematically, the constraint is of the form,

$$a_j x_j + h_k(x_i) \leq b_k, \tag{S}$$

where, $a_j \neq 0$ and $j \neq i$. A nonlinear constraint with more than one term in its linear part can be transformed into this structure by replacing the entire linear part using an auxiliary variable. This univariate structure appears in 126 out of 334 instances in test set *TS*. Problem classes with this structure are listed in Table 1. In 123 of these instances, all the nonlinear constraints have this structure. Three instances, `ex1223a` and two of `synthes*`, have a few other constraints without this structure. We refer to the set of these 126 instances as TS_1 and the set of remaining 208 instances in *TS* as TS_2. The structure (S) is also exploited in [29] for building initial relaxation in outer approximation algorithms. They select points at regular intervals along x_i.

The feasible region of (S) can be visualized in the two-dimensional space of x_i and x_j variables. It is easy to see that a linearization generated at any point (x_i, x_j) in

Table 1 Name of classes and number of instances (#) with the univariate structure (S) in a class. * following a name denotes a collection of instances in a class

Name	#	Name	#	Name	#
cvxnonsep_normcon*r	3	fo8*	6	procurement2mot	1
cvxnonsep_nsig*r	3	fo9*	6	rsyn*m	24
cvxnonsep_pcon*r	3	m*	8	sssd*	13
ex1223a	1	no7*	5	syn*m	24
flay*	10	nvs03	1	synthes*	2
fo7*	7	o7*	9	**Total**	**126**

the plane touches the constraint boundary at some point. We utilize this simple fact in the first two schemes.

3.1.1 Root linearization scheme 1 (RS1)

Given a nonlinear constraint with the univariate structure (S), this iterative scheme selects a point in each iteration for generating a linearization until the violation of the nonlinear constraint at all points in the updated $\bar{P}_k$ is less than a desired value $\tilde{T}_k$.

The scheme starts by generating linearizations at points $x^L = (l_i, (b_k - h_k(l_i))/a_j)$ and $x^U = (u_i, (b_k - h_k(u_i))/a_j)$, where l_i and u_i are the lower and upper bounds, respectively, on x_i. Both x^L and x^U lie on the boundary of the feasible region of (S). Let us add to $\bar{P}_k$ two linearizations $L(x^L)$ and $L(x^U)$ at these points. Amongst all points in the updated $\bar{P}_k$, the violation of the constraint (S) is maximum at the point of intersection, x^I, of $L(x^L)$ and $L(x^U)$. Let E_k be the set of extreme points of $\bar{P}_k$. At any point $x^l \in E_k$, let $v(x^l)$ be the violation of the nonlinear constraint defined as $v(x^l) = \max\{a_j x_j^l + h_k(x_i^l) - b_k, 0\}$, where x_i^l and x_j^l are the values of variables x_i and x_j in x^l. In each iteration, candidate points for generating a new linearization are those points $x^l \in E_k$ for which $v(x^l) \geq \tilde{T}_k$, amongst whom the most violated point is selected. Figure 2 shows a pictorial depiction of this scheme and Algorithm 1 presents the pseudocode for this scheme.

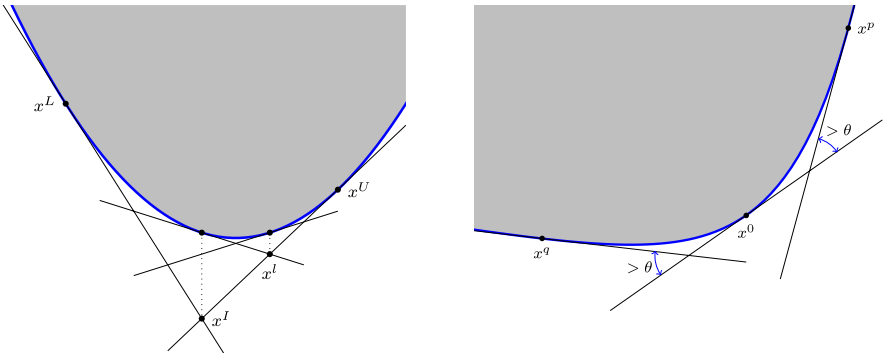

Fig. 2 Pictorial depiction of linearization scheme RS1 (left) and RS2 (right)

Algorithm 1: Root linearization scheme RS1.

Input: Nonlinear constraint indexed k with structure (S), a scalar K and initial $\bar{P}_k$.

1 Compute points x^L and x^U, generate linearizations $L(x^L)$ and $L(x^U)$ for the nonlinear constraint at these points, and add to $\bar{P}_k$.
2 Compute intersection point, x^I, of $L(x^L)$ and $L(x^U)$, and threshold value $\tilde{T}_k$.
3 Construct set $E_k = \{x^L, x^I, x^U\}$ of the extreme points of $\bar{P}_k$.
4 **while** $(\max_{x^l \in E_k}\{v(x^l)\} \geq \tilde{T}_k)$ **do**
5 Select $x^p \in E_k$ with the maximum violation value, generate linearization for the nonlinear constraint at x^p and add to $\bar{P}_k$.
6 Update set E_k by adding newly generated extreme points.

We compare the default implementation of QG in Minotaur, which we refer to as *qg* to that of *qg* with RS1 (denoted as *qgrs1*). Threshold $\tilde{T}_k$ is set to be a fraction K of b_k, if $b_k \neq 0$, otherwise of $v(x^I)$. We tried four different values of K : 0.02, 0.05, 0.10, 0.20. In case any of the bounds, l_i or u_i, on variable x_i is not known, we take $l_i = x_i^0 - 50$ and $u_i = x_i^0 + 50$, respectively. Table 2 shows the impact of this scheme on the overall solution time, size of the tree in terms of the number of nodes processed, and the Euclidean distance of the optimal solution ($\bar{x}$) of the root LP from the feasible region of (R). The following nonlinear program is solved for computing this distance.

$$\begin{aligned} \min_x \ & ||x - \bar{x}||_2 \\ \text{s.t. } & g(x) \leq b, \\ & x \in X. \end{aligned} \tag{NLP-D}$$

Problem (NLP-D) differs from (R) only in the objective function.

Each row of the top table in Table 2 corresponds to a parameter setting (K in this case). The column '# Solved by' lists the number of instances solved to optimality within the time limit by the proposed method and by both the reference solver (*qg* in this case) as well as the proposed method (under the column 'Both'). The first column under the headings 'Time' and 'Nodes' shows the shifted geometric mean (SGM) of these measures reported by the reference solver for the instances in the column 'Both'. The second column under these headings show the relative SGM ('Rel.') of the proposed method for the same instances using the setting corresponding to the row. Similar statistics for the distance measure are computed, but over all instances, not just for those solved within the time limit. The relative SGM of a measure is computed as the ratio of the SGM value of the proposed scheme to the SGM value of the reference solver (*qg* in this section). If this ratio, say r, is less than one, it implies that the proposed solver has performed better than the reference solver. More specifically, the proposed solver has shown $(1 - r) \times 100\%$ improvement over the reference solver on the considered performance measure. For example, *qgrs1* with $K = 0.20$ is on an average 11% faster and showed an improvement of about 15% and 81% in the number of nodes processed and distance, respectively, over default *qg* on the set of 111 instances that were solved by both *qg* and *qgrs1*. We used a shift of 10 for calculating SGM of time and distance, and 100 for the number of nodes processed.

Table 2 (Top) Comparison of *qg* and *qgrs1* for different values of K on test set TS_1. *qg* could solve 113 instances in the time limit. (Bottom) Performance break-up of *qgrs1* with $K = 0.20$ over instances of varying difficulty in TS_1

K	# Solved by		Time		Nodes		Distance	
	qgrs1	Both	*qg*	Rel.	*qg*	Rel.	*qg*	Rel.
0.02	113	111	31.66	0.93	6.9e3	0.79	1.35	0.02
0.05	113	111	31.38	0.86	6.9e3	0.77	1.36	0.05
0.10	113	111	31.54	0.91	6.9e3	0.86	1.37	0.08
0.20	115	111	31.54	0.89	6.9e3	0.85	1.37	0.19

Time	# Solved by	Time		Nodes		Distance	
	Both	*qg*	Rel.	*qg*	Rel.	*qg*	Rel.
> 0	111	31.54	0.89	6.9e3	0.85	1.37	0.19
> 10	53	164.82	0.86	1.5e5	0.88	1.19	0.12
> 100	31	453.82	0.79	4.7e5	0.79	0.89	0.09
> 500	14	1133.83	0.87	1.2e6	0.86	0.68	0.06

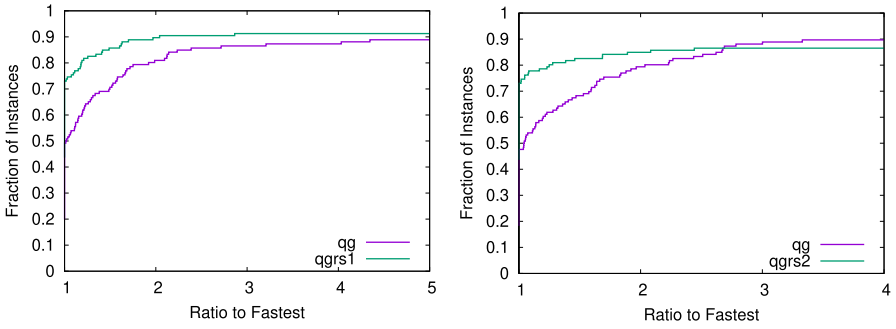

Fig. 3 Performance profiles comparing solution times of *qg* and *qgrs1* with $K = 0.20$ (on left) and of *qg* and *qgrs2* with $\theta = 5$ (on right) on instances in TS_1

Our computational results report modest improvements in all the considered measures under all the settings. We choose $K = 0.20$ as the default setting for this scheme as it solved 2 instances more than *qg* and resulted in about 11% improvement in solution times. The break-up of performance over instances of varying difficulty using the best setting is also included in Table 2 (bottom). Each row corresponds to the instances solved by both *qg* and *qgrs1*, but for which at least one of them took more than the specified time. For example, 31 instances were solved to optimality by both *qg* and *qgrs1*, and for each of these instances, at least one of the two solvers took more than 100 seconds. We observed that *qgrs1* is more effective for 'difficult' problems, especially those corresponding to row 3 in the table on the bottom. Similar tables have been used in the rest of the paper as well. We use performance profiles [21] that graphically demonstrate the relative performance of different solvers for a particular performance measure over a given set of instances. Let S be a set of solvers to be compared, I be a

given set of instances, and $t_{i,s}$ be the solution time of instance $i \in I$ by solver s. The performance ratio $r_{i,s}$ of solver s on instance i compared to the best solver for this instance is given by

$$r_{i,s} = \frac{t_{i,s}}{\min\limits_{j \in S} t_{i,j}},$$

and $\rho_s(\tau) : \mathbb{R} \to [0, 1]$, a cumulative distribution function for the performance ratio of solver s, is defined as

$$\rho_s(\tau) = \frac{|i \in I : r_{i,s} \leq \tau|}{|I|}.$$

$\rho_s(\tau)$ is a nondecreasing function indicating that solver s is at most τ times slower than the best solver on an instance. In particular, the value $\rho_s(1)$ gives the fraction of the instances on which a solver s performs the best. Figure 3 (left) shows the performance profiles of *qg* and *qgrs1* with $K = 0.20$ (proposed best setting of RS1) using the solution times of the instances in test set TS_1. A close look at Fig. 3 (left) shows that $\rho_{qgrs1}(1) = 0.45$, which means that *qgrs1* has performed better on about 45% of the total instances in TS_1. Next, we see from the profile of *qg* that *qg* could solve about 90% of the total instances. Moreover, the value of $\rho_{qg}(2)$ is about 0.8, which means that for about 80% of the instances, *qg* is at most two times slower than *qgrs1*. This also means that on the remaining 10% of the instances *qg* is at least two times slower (or *qgrs1* is at least two times faster). We use similar profiles for reporting the results of the other schemes in this section.

3.1.2 Root linearization scheme 2 (RS2)

Given a nonlinear constraint with univariate structure (S), this scheme iteratively selects points in a way that the successively generated linearization constraints differ in slope by at least a specific threshold value. Like RS1, the feasible region of (S) can be seen as a two-dimensional region in the space of x_i and x_j variables. We start at x^0, an optimal solution of (R). First, x_i^0 is gradually increased using step size δ and variable x_j is determined. If the slope of the linearization at this point differs from the slope of the previously accepted linearization by θ, it is added to the linear relaxation. Otherwise the step size δ is doubled. This process is repeated until x_i^0 exceeds $\min\{u_i, x_i^0 + \Delta\}$ for a scalar parameter $\Delta > 0$. A similar search is carried out in the opposite direction until x_i^0 falls below $\max\{l_i, x_i^0 - \Delta\}$. Figure 2 gives a pictorial description of the scheme and Algorithm 2 presents pseudocode of RS2 along the direction $-e_i$.

Algorithm 2: Root linearization scheme RS2 for the direction $-e_i$.

Input: Nonlinear constraint k with structure (S), x^0, θ, Δ, δ, l_i, u_i, and iteration $p = 1$.

if $x_i^0 - \max\{l_i, x_i^0 - \Delta\} < 1$ **then**
  Set $\delta = x_i^0 - \max\{l_i, x_i^0 - \Delta\}$
Set $x_i^1 = x_i^0 - \delta$ and $x_j^1 = (b_k - h_k(x_i^1))/a_j$.
while $x_i^p \geq max\{l_i, x_i^0 - \Delta\}$ **do**
  Compute angle, α, between the linearization at x^p and the last generated linearization.
  if $\alpha \geq \theta$ **then**
    Generate linearization at x^p.
  else
    Set $\delta \leftarrow 2\delta$.
  Set $p = p + 1, x_i^p = x_i^{p-1} - \delta$ and $x_j^p = (b_k - h_k(x_i^p))/a_j$.

Computational performance of *qg* with linearization scheme RS2 (*qgrs2*) on TS_1 is presented in Table 3. We used four values of $\theta = 2, 5, 10, 20$ with $\Delta = 10$, and $\delta = 0.5$. In our experiments, we observed improvements in solution time and tree-size for the first two settings. Quality of the relaxation improved for all the considered θ values - more for smaller values because more linearizations were added, implying a tighter, but larger LP. Although more instances than *qg* and other settings were solved with $\theta = 10$, it resulted in poor solution times. *qgrs2* with $\theta = 5$ solved two instances less than *qg* but resulted in better performance on 109 instances that were solved by both *qg* and *qgrs2*. Table 3 shows the break-up of its performance over instances of varying difficulty. As we increase the value of θ the number of linearizations added to the root relaxation decreases. As we increase the value of θ, the number of linearizations added to the root relaxation decreases. When very few linearizations are added, the root relaxations obtained in *qgrs2* and *qg* are almost identical. Therefore, their performance does not vary much, as indicated in the third column of Table 3 by the relative time measure close to 1 for $\theta = 10, 20$. The performance profiles in Fig. 3 (right) compare the solution times of instances in TS_1 when solved using *qg* and *qgrs2* with $\theta = 5$. We observe that on about 45% of the instances, *qgrs2* is faster than *qg* and at least two times faster on about 5% of the instances.

Time taken within the schemes *qgrs1* and *qgrs2* is negligible (less than 0.5s) in comparison to the total solution time for all the considered instances. The next three schemes are applicable to a general problem of the form (P).

3.1.3 Root linearization scheme 3 (RS3)

This scheme finds linearization points near an optimal solution of an LP relaxation of (R). First the LP relaxation (problem (RM) with integrality relaxed) is solved and an optimal solution, $\bar{x}$, is obtained. If the solution violates any nonlinear constraint, a line-search is performed between $\bar{x}$ and x^C to find a point at the boundary of the feasible region of (R). x^C is chosen to be a point inside the

Table 3 (Top) Comparison of *qg* and *qgrs2* for different values of θ on test set TS_1. *qg* could solve 113 instances. (Bottom) Break-up of performance over instances of varying difficulty for *qgrs2* with $\theta = 5$

θ	# Solved by		Time		Nodes		Distance	
	qgrs2	Both	*qg*	Rel.	*qg*	Rel.	*qg*	Rel.
2	112	110	30.44	0.95	6.6e3	0.88	1.37	0.11
5	111	109	28.64	0.87	6.2e3	0.86	1.38	0.17
10	115	112	33.01	1.00	7.3e3	1.00	1.35	0.50
20	113	112	33.01	1.01	7.3e3	1.03	1.35	0.73

Time	# Solved by	Time		Nodes		Distance	
	Both	*qg*	Rel.	*qg*	Rel.	*qg*	Rel.
> 0	109	28.64	0.87	6.2e3	0.86	1.38	0.17
> 10	51	147.57	0.81	1.3e5	0.82	1.20	0.10
> 100	28	432.08	0.74	4.4e5	0.72	0.92	0.07
> 500	13	964.24	0.85	1.0e6	0.83	0.65	0.05

feasible region of (R). The boundary point is used to generate new linearizations. The updated LP is solved again and the process is continued. The point x^C remains the same at every iteration. We stop when the LP solution is feasible to (R) or a preset number (k_{max}) of LPs have been solved.

To obtain interior point x^C, we solve the following nonlinear problem (NLP-I). All the nonlinear inequalities in (R) are modified using an auxiliary variable ν, which also forms the objective of this new NLP. All the other (linear) constraints remain unchanged.

$$\begin{aligned} \min_{x,\nu} \ & \nu \\ \text{s.t. } & g(x) \leq b + \nu, \\ & x \in X, \\ & \nu \leq 0, x \in \mathbb{R}^n. \end{aligned} \tag{NLP-I}$$

Let the optimal solution of (NLP-I) be $(\tilde{\nu}, \tilde{x})$. If $\tilde{\nu} < 0$, then we set $x^C = \tilde{x}$. If $\tilde{\nu} = 0$, then there does not exist any point in the feasible region of (R) at which all the nonlinear constraints are inactive. In this case, we simply generate linearizations to nonlinear constraints that are active at $\tilde{x}$, and terminate the scheme. If (NLP-I) is unbounded, then we add ν to the linear inequalities in the same way as the nonlinear constraints and re-solve.

Algorithm 3 presents the pseudocode for this scheme. This scheme is similar to root LP generation in the ESH algorithm in [33], but differs in the formulation of initial root LP and the nonlinear problem (NLP-I). Unlike [33], our initial root LP is obtained by linearizing nonlinear constraints at x^0, and we also consider linear equalities to find the required interior point, thus ensuring x^C lies in the feasible region of (R). Out of the total 334 instances in test set *TS*, 67

have nonlinearity only in the objective. The remaining 267 resulted in an optimal solution with $\tilde{v} < 0$. Our computational investigations indicate that the choice of interior point plays an important role in determining the quality of linearizations generated. We experimented first with x^C as obtained from solving (NLP-I). Next, we used the center of the line segment between x^C and x^0 as the required interior point, which also lies in the interior of the feasible region of (R). Interior point obtained using the latter way resulted in a better performance.

For 67 instances with nonlinearity only in the objective function (all these instances lie in TS_2), root LP solution is also feasible to the problem (R). We add objective linearization directly at the LP solution obtained in every iteration. The algorithm, in this case, terminates when the current LP solution is the same as the previous solution or when we exhaust a prefixed number of iterations, k_{max}.

Algorithm 3: Root linearization scheme RS3.

Input: An interior point x^C, maximum iteration limit k_{max}, the initial root LP and its solution $\bar{x}$.

1 Set the iteration count $k = 1$ and $\bar{x}^1 = \bar{x}$.
2 **while** ($k \leq k_{max}$ *and* $\bar{x} \notin$ *feasible region of* (R)) **do**
3 Find $0 \leq \lambda \leq 1$ such that $x^B = \lambda x^C + (1-\lambda)\bar{x}^k$ lies on the boundary of (R).
4 Add linearizations of all the nonlinear constraints active at x^B.
5 Set $k = k + 1$, solve the resulting LP and let $\bar{x}^k$ be its optimal solution.

We compare *qg* and *qg* with scheme RS3 (*qgrs3*) using $k_{max} = 5, 10, 20, 40$. Tables 4 and 5 report values for different performance metrics on TS_1 and TS_2 respectively. The number of LPs to be solved and the number of linearizations added at the root node of the search tree increase or remain the same as k_{max} is increased.

Table 4 (Top) Comparison of *qg* and *qgrs3* for different values of k_{max} on test set TS_1. *qg* could solve 113 instances. (Bottom) Break-up of results over instances of varying difficulty with the best setting $k_{max} = 10$

k_{max}	# Solved by		Time		Nodes		Distance	
	qgrs3	Both	*qg*	Rel.	*qg*	Rel.	*qg*	Rel.
5	113	110	29.94	0.99	6.5e3	1.03	1.38	0.26
10	113	111	31.54	0.95	6.9e3	0.98	1.37	0.12
20	113	111	31.54	1.01	6.9e3	1.01	1.37	0.05
40	112	111	31.66	1.06	6.9e3	1.06	1.35	0.01

Time	# Solved by	Time		Nodes		Distance	
	Both	*qg*	Rel.	*qg*	Rel.	*qg*	Rel.
> 0	111	31.54	0.95	6.9e3	0.98	1.37	0.12
> 10	54	156.81	0.92	1.4e5	0.98	1.17	0.05
> 100	31	451.61	0.84	4.7e5	0.88	0.89	0.02
> 500	17	933.18	0.81	1.0e6	0.83	0.64	0.02

Table 5 (Top) Comparison of *qg* and *qgrs3* for different values of k_{max} on test set TS_2. *qg* could solve 179 instances. (Bottom) Break-up of results over instances of varying difficulty with the best setting $k_{max} = 40$

k_{max}	# Solved by		Time		Nodes		Distance	
	qgrs3	Both	*qg*	Rel.	*qg*	Rel.	*qg*	Rel.
5	180	177	11.86	0.99	1.0e3	1.00	5.73	0.48
10	179	176	11.33	0.95	1.0e3	0.96	5.78	0.28
20	179	177	11.86	0.96	1.0e3	0.93	5.73	0.19
40	179	177	11.86	0.92	1.0e3	0.88	5.73	0.07

Time	# Solved by	Time		Nodes		Distance	
	Both	*qg*	Rel.	*qg*	Rel.	*qg*	Rel.
> 0	177	11.86	0.92	1.0e3	0.88	5.73	0.07
> 10	64	56.41	0.88	1.0e4	0.82	17.5	0.06
> 100	18	336.23	0.93	1.4e4	0.89	4.61	0.08
> 500	7	1548.07	0.90	3.8e4	0.95	0.56	0.19

If this is the case, the size of root relaxation also increases with k_{max}, and it may take longer to solve. If the scheme RS3 terminates due to other criteria before reaching its limit of k_{max}, then the root relaxation and hence the solution time will be the same for all the values greater than or equal to k_{max}. The time taken in this scheme is a very small fraction of the total solution time in all the considered instances. The maximum time taken was close to 2s for instances with a large number of variables.

We observed only small improvements in the performance metrics over the set TS_1 and reasonable improvements for TS_2. We obtained an improvement of about 5% on TS_1 and of 8% on TS_2 in solution times. Tables 4 and 5 (bottom ones) provide a break-up of performance for instances in TS_1 and TS_2, respectively, from the best settings on these sets. Larger improvements in solution times are seen for more difficult and structured instances (Table 4, last row in the bottom table). However, we can not predict whether an instance is 'easy' or 'difficult' before solving it. A comparison of solution times of *qg* and *qgrs3* on TS_1 and TS_2 are presented in Fig. 4 (on the left and the right, respectively). On both the test sets, *qgrs3* is seen faster on about 35% of the instances. Overall, *qgrs3* is slower than *qgrs1* and *qgrs2* on TS_1.

3.1.4 Root linearization scheme 4 (RS4)

In this scheme we search for linearization points by exploring several 'well spread' directions. Starting from an interior point of the feasible region of (R), we move along each chosen direction until the boundary of the feasible region of (R) is reached. We add linearizations to all the nonlinear constraints that are active at the obtained boundary point. The interior point is computed in the same way as in RS3. For search directions, we use positive and negative standard basis which consists of directions of the form $\{e_j, -e_j\}$, $\forall j \in D$, where D is the set of indices of variables

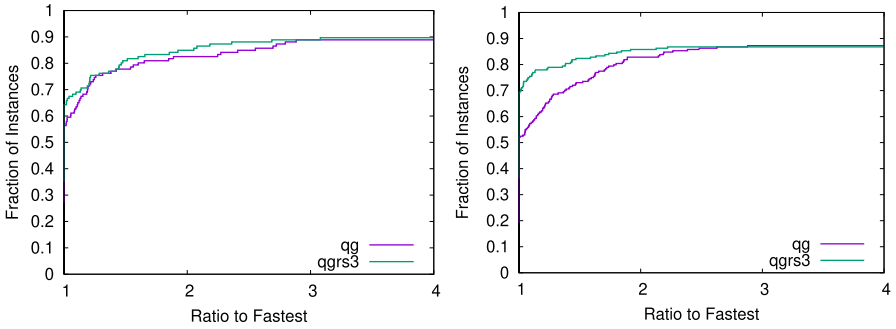

Fig. 4 Performance profiles comparing solution times of *qg* and *qgrs3* with $k_{max} = 10$ on instances in TS_1 (on left) and with $k_{max} = 40$ on instances in TS_2 (on right)

that appear in the nonlinear part of some constraint or objective and e_j is the j^{th} unit vector. This means that there are at most $2|D|$ directions and points for linearizations.

For problems with nonlinearity only in the objective function, this scheme is changed slightly. In the problem (NLP-I) for finding an interior point, linear inequalities are modified in the same way as nonlinear inequalities. If $\tilde{v} < 0$, then the scheme is same as for the problems with nonlinear constraints with the only difference that in the place of nonlinear constraints, linear constraints are used. In the rare case of $\tilde{v} = 0$ or if there exists a linear equality constraint, the point $\tilde{x}$ lies on the boundary of the feasible region of (R). In this case, we consider the four equidistant points on the line segment between $\tilde{x}$ and x^0. We generate linearizations at these four points that also lie on the boundary of the feasible region of (R). A similar step is performed along the opposite direction $d = \tilde{x} - x^0$. Starting from $\tilde{x}$, we consider four equidistant points on the line segment between $\tilde{x}$ and $2\tilde{x} - x^0$. Out of these four points, the ones which are feasible to (R) are selected for generating linearizations.

We observed that in set TS_2, many instances in the class of problems such as ibs2, squfl0*, unitcommit_200_100*, watercontamination*, etc., have a large number of variables in their nonlinear part resulting in a large number of elements in the set D. For such problems, we restrict the size of set D, thus limiting the amount of time spent in this scheme by searching along fewer directions.

In our runs, we limit the size of D to a maximum of 300 selecting only the first 300 directions. First, we chose x^C as defined in RS3 scheme (Sect. 3.1.3) as the interior point and referred to this setting as FC. Then, we used the mid-point of the line segment joining x^C and x^0 as the interior point; this setting is termed as MC. Results from *qg* with RS4 (*qgrs4*) on TS_1 and TS_2 using these two settings are shown in Tables 6 and 7 respectively. The time taken within this scheme is again a very small fraction of the total solution time, most of which is spent in solving the nonlinear problem (NLP-I) for finding the interior point.

On both the test sets, setting MC has performed better. On TS_1, *qgrs4* solved same number of instances as *qg*, but resulted in an improvement of about 12% in the solution times. Overall, this scheme is inferior to *qgrs1*, but better than both *qgrs2* and *qgrs3* on this test set. On TS_2, *qgrs4* solved two instances fewer than *qg*,

Table 6 (Top) Comparison of *qg* and *qgrs4* on test set TS_1. *qg* could solve 113 instances. (Bottom) Break-up of performance over instances of varying difficulty with best setting MC

Setting	# Solved by		Time		Nodes		Distance	
	qgrs4	Both	*qg*	Rel.	*qg*	Rel.	*qg*	Rel.
MC	113	111	31.54	0.88	6.9e3	0.89	1.37	0.61
FC	113	110	29.94	0.92	6.5e3	0.92	1.38	0.76

Time	# Solved by	Time		Nodes		Distance	
	Both	*qg*	Rel.	*qg*	Rel.	*qg*	Rel.
> 0	111	31.54	0.88	6.9e3	0.89	1.37	0.61
> 10	54	156.81	0.82	1.4e5	0.88	1.17	0.47
> 100	33	414.91	0.75	4.3e5	0.79	0.86	0.46
> 500	14	1133.83	0.73	1.2e6	0.77	0.68	0.21

Table 7 (Top) Comparison of *qg* and *qgrs4* on test set TS_2. *qg* could solve 179 instances. (Bottom) Break-up of performance over instances of varying difficulty with best setting MC

Setting	# Solved by		Time		Nodes		Distance	
	qgrs4	Both	*qg*	Rel.	*qg*	Rel.	*qg*	Rel.
MC	177	177	11.86	0.94	1.0e3	0.93	5.73	0.59
FC	178	177	11.86	1.00	1.0e3	0.96	5.73	0.59

Time	# Solved by	Time		Nodes		Distance	
	Both	*qg*	Rel.	*qg*	Rel.	*qg*	Rel.
> 0	177	11.86	0.94	1.0e3	0.93	5.73	0.59
> 10	63	57.50	0.93	1.1e4	0.94	7.62	0.50
> 100	16	416.56	0.86	2.3e4	0.88	7.56	0.71
> 500	7	1548.07	0.80	3.8e4	0.86	0.56	1.00

but on 177 instances that were solved by both, it showed an improvement of about 6%. Although, *qgrs4* has solved two instances fewer than *qgrs3*, it seems to have performed better on 'difficult' instances (rows corresponding to time > 500 in the respective tables). The performance profiles in Fig. 5 (left) indicate that *qgrs4* is faster than *qg* on about 40% of the total instances in TS_1 and Fig. 5 (right) shows that *qgrs4* is faster than *qg* on about 40% of the total instances in TS_2.

3.1.5 Root linearization scheme 5 (RS5)

This scheme selects points for linearization in a neighborhood of x^0, the optimal solution obtained by solving (R). Starting from x^0, we move in different directions to find suitable points. We consider two sets of directions. For the first set, we select

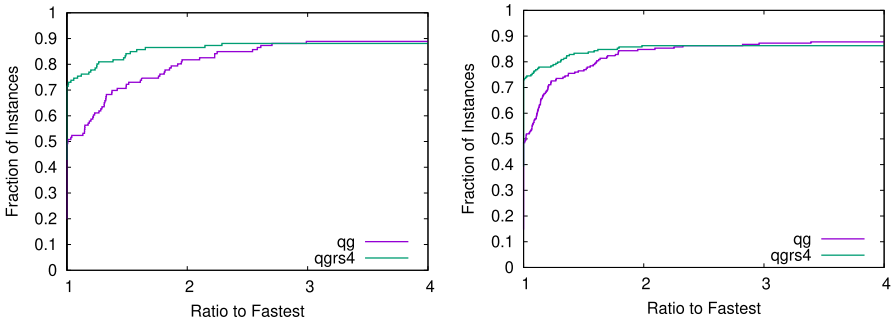

Fig. 5 Performance profiles comparing solution times of *qg* and *qgrs4* with MC on instances in TS_1 (left) and TS_2 (right)

affinely independent points on the hyperplane passing through x^0 and whose normal is $(x^C - x^0)$, where x^C is an interior point like in RS3 (Sect. 3.1.3). Let this hyperplane be denoted by $a^T x = r$, where $a = x^C - x^0$, $j = 1, \ldots, n$ and $r = (x^C - x^0)^T x^0$. Let j be any index such that $a_j \neq 0$, and define a set of n affinely independent points x^i on this hyperplane as

$$x^i = \begin{cases} (r/a_i)e_i, & \text{if } a_i \neq 0, \\ (r/a_j)e_j + e_i, & \text{otherwise.} \end{cases} \tag{2}$$

Let DS_1 be the set of $(n-1)$ linearly independent directions, $x^i - x^1, i = 2, \ldots, n$. Each of these directions has at most two nonzero components.

For each direction d from the set DS_1, we search iteratively along d starting from x^0. At iteration l, we obtain a point $\bar{x}^l = \bar{x}^{l-1} + \delta d$, where δ is a positive step size and $\bar{x}^0 = x^0$. Then, starting from x^C, we perform a line search along direction $(\bar{x}^l - x^C)$ for finding a point x^B on the boundary of the feasible region of (R). For every nonlinear constraint active at x^B, we compute the angle between the normals of the linearization drawn at x^B and the previous linearization added to this nonlinear constraint. If this angle is more than a specified threshold θ (in degrees), then we add the linearization generated at x^B to the relaxation. If the objective is also nonlinear, we add an objective linearization at x^B using the same criterion of slope difference. If no linearizations are added at the current point x^B, then we double the step size δ and repeat the search. The search terminates when any component of the point $\bar{x}^l$ violates its bound (lower or upper). This process is repeated for every direction $d \in DS_1$ and also its negative. The whole procedure was tried on another set of directions, DS_2, as a variant of the above method. For each $d \in DS_1$, we replace its negative components by -1 and positive components by 1 to get a new direction. All these $n-1$ directions constitute DS_2. Rest of the procedure remains identical.

In order to choose an initial step size δ along a direction d, we consider the Hessian of the Lagrangian, H, at x^0. If the absolute value of $d^T H d$ is below a threshold, we take a step size δ_l, otherwise a smaller step size δ_s is chosen. For problems with nonlinearity only in the objective function, this scheme is modified in the same way as in RS4. However, unlike scheme RS4, if $\tilde{\nu} = 0$ or if there exists a linear equality

Table 8 (Top) Comparison of *qg* and *qgrs5* with FC-2 for different values of θ on test set TS_1. *qg* could solve 113 instances. (Bottom) Break-up of performance over instances of varying difficulty with the best setting $\theta = 2$

θ	# Solved by		Time		Nodes		Distance	
	qgrs5	Both	*qg*	Rel.	*qg*	Rel.	*qg*	Rel.
2	115	113	34.71	0.93	7.8e3	0.94	1.34	0.69
5	113	112	33.05	0.92	7.4e3	0.96	1.35	0.82
10	112	112	33.05	0.94	7.4e3	0.99	1.35	0.84
20	113	113	34.71	0.95	7.8e3	0.98	1.34	0.93

Time	# Solved by	Time		Nodes		Distance	
	Both	*qg*	Rel.	*qg*	Rel.	*qg*	Rel.
> 0	113	34.71	0.93	7.8e3	0.94	1.34	0.69
> 10	55	183.02	0.88	1.7e5	0.91	1.14	0.64
> 100	33	505.64	0.84	5.5e5	0.86	0.84	0.67
> 500	16	1261.25	0.88	1.4e6	0.89	0.60	0.32

constraint, then we consider points at an interval of δ_s on the line segment between the points x^C and x^0.

In our numerical experiments using *qg* with RS5 (*qgrs5*), we first used x^C (referred to as FC) and then modified it as in *qgrs4* (denoted as MC). Along with the two proposed set of directions, DS_1 and DS_2, we obtained four settings: FC-1, MC-1, FC-2, MC-2; for example, FC-1 corresponds to the setting in which interior point is chosen as *FC* and search directions are from DS_1. For each setting, we used four values for parameter $\theta = 2, 5, 10, 20$, $\delta_s = 0.25$, and $\delta_l = 1$. Out of the four settings, FC-2 with $\theta = 2$ exhibited the best results on both the test sets and are presented in Tables 8 and 9. On TS_1, *qgrs5* with this setting solved 2 instances more than *qg* and exhibited an improvement of about 7% in solution times. Overall, on TS_1, *qgrs5* is inferior to all the previous schemes except *qgrs3* in terms of solution times, but is better than all except *qgrs1* in terms of number of instances solved. On TS_2, it solved one instance more than *qg* and provided an improvement of about 12% in solution times. It also provided better solution times than *qgrs3* and *qgrs4*. Like *qgrs3* and *qgrs4*, most of the time taken by *qgrs5* is spent in solving the nonlinear problem (NLP-I) for finding the interior point. Profiles in Fig. 6 compare solution times of *qg* and *qgrs5* on TS_1 and TS_2. These results show that *qgrs5* is faster than *qg* on about 45% of the total instances in TS_1, and on about 40% of the total instances in TS_2.

3.2 Adding linearization constraints at other nodes

We now consider schemes for nodes (other than the root) that yield a fractional optimal solution in the branch-and-bound tree. Two main decisions in the design of these schemes are: (a) whether additional linearization constraints should be

Table 9 (Top) Comparison of *qg* and *qgrs5* with FC-2 for different values of θ on test set TS_2. *qg* could solve 179 instances. (Bottom) Break-up of performance over instances of varying difficulty with the best setting $\theta = 2$

θ	# Solved by		Time		Nodes		Distance	
	qgrs5	Both	*qg*	Rel.	*qg*	Rel.	*qg*	Rel.
2	180	177	11.86	0.88	1.0e3	0.88	5.73	0.27
5	180	177	11.86	0.95	1.0e3	0.92	5.73	0.56
10	179	177	11.86	0.97	1.0e3	0.94	5.73	0.88
20	178	176	11.33	0.98	1.0e3	0.98	5.78	0.90

Time	# Solved by	Time		Nodes		Distance	
	Both	*qg*	Rel.	*qg*	Rel.	*qg*	Rel.
> 0	177	11.86	0.88	1.0e3	0.88	5.73	0.27
> 10	62	59.15	0.83	1.1e4	0.82	18.08	0.15
> 100	15	451.28	0.81	2.0e4	0.88	4.55	0.34
> 500	7	1548.07	0.87	3.8e4	0.93	0.56	1.11

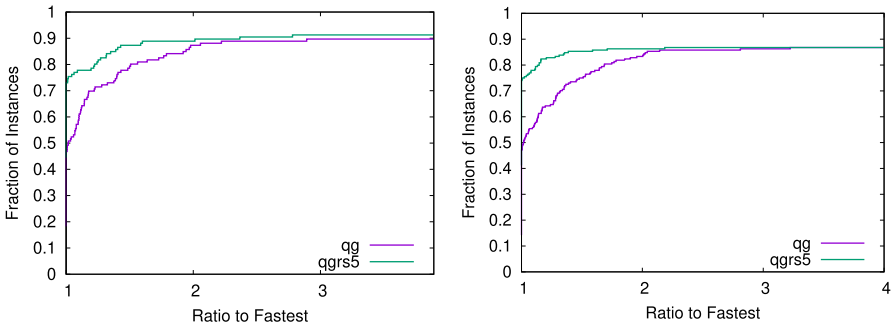

Fig. 6 Performance profiles comparing solution times of *qg* and *qgrs5* with FC-2 and $\theta = 2$ on instances in TS_1 (left) and TS_2 (right)

added at a given node, and (b) how to determine points for generating linearization constraints.

3.2.1 Node linearization scheme 1 (NS1)

Let x' and z' be the optimal solution and corresponding optimal value obtained by solving the LP relaxation at a node. For a nonlinear constraint, $g_k(x) \leq b_k$, we assign a violation based score $V^k = v_k/|b_k|$, if $b_k \neq 0$, and $V^k = v_k$ otherwise, with $v_k = max\{0, g_k(x') - b_k\}$. For a nonlinear objective, $f(x)$, score V^o is defined as $V^o = v_o/|z'|$, if $z' \neq 0$, and $V^o = v_o$ otherwise, with $v_o = max\{0, f(x') - z'\}$. If the score of a nonlinear constraint is more than a preset threshold value τ, then we

generate linearizations at the node. To avoid adding too many cuts, this scheme is applied only up to a certain depth D in the branch-and-bound tree.

We employ the following two methods for finding points for generating linearizations for problems that have at least one nonlinear constraint. The first method is based on the extended cutting plane technique [57], hence we refer to it as the ECP method. Here, we generate linearizations at x' to all nonlinear constraints whose score $V^k \geq \tau$. If the objective is nonlinear, then we add a linearization to the objective at x' if $V^o \geq \tilde{K}$. Here, $\tilde{K}$ is initialized with $v_r/|\bar{z}|$, if $\bar{z} \neq 0$, and v_r otherwise, where $v_r = max\{0, f(\bar{x}) - \bar{z}\}$, and $\bar{x}$ and $\bar{z}$ denote an optimal solution and corresponding optimal value to the root LP relaxation. If $\tilde{K} < 0.5$, then we double the value of $\tilde{K}$.

The second method is based on line-search and we refer to it as the LS method. Starting from an interior point $\tilde{x}$ in the feasible region of (R), we search along the direction $x' - \tilde{x}$ for a point on the boundary of the feasible region of (R). Then we generate linearizations at this boundary point to all the active nonlinear constraints. This method ensures that all the linearizations are tight. The chosen interior point $\tilde{x}$ is the mid-point of the line segment joining x^C (an interior point obtained as in Sect. 3.1.3) and x^0. For problems with a nonlinear objective also, we add a linearization at the obtained boundary point if the criteria mentioned in ECP are met.

For problems that have nonlinearity only in the objective function, a node is selected for adding linearization if $V^o \geq \tilde{K}$, where $\tilde{K}$ is initialized in the same way as above. If $\tilde{K} > 1000$, then we reduce depth D by half, and if $\tilde{K} < 0.5$, we double the value of $\tilde{K}$ and D. For these problems we employ only ECP method. This treatment to the problems with nonlinearity only in the objective remains the same in the following two schemes, NS2 and NS3, as well.

Using *qg* with scheme NS1 (*qgns1*), we experimented with four values of $\tau : \{0.75, 1, 1.5, 2\}$, $D = 10$ for problems with nonlinear constraints, and $D = 5$ for problems with nonlinearity only in the objective. This scheme with both the methods have shown improvements in solution time and the number of nodes processed. We obtained better results with LS method than ECP on both the test sets. Results for TS_1 and TS_2 are reported in Tables 10 and 11 respectively. On TS_2, although *qgns1* solved one instance less than *qg*, fair improvements are seen in solution times. Best results are obtained using $\tau = 2$, with an improvement of about 7% and 11% in solution times on TS_1 and TS_2, respectively. The solution times of *qg* and *qgns1* on TS_1 and TS_2 are compared in Fig. 7. In these performance profiles, *qgns1* is reported to be faster on about 60% of the total instances in TS_1, and on about 45% of the total instances in TS_2. On TS_1, this scheme is inferior to all the root schemes in terms of solution times, but comparable to *qgrs1* and *qgrs5* in terms of the number of instances solved. On TS_2, this scheme has performed better than root schemes *qgrs3* and *qgrs4*, but inferior to *qgrs5*.

3.2.2 Node linearization scheme 2 (NS2)

This scheme is similar to NS1 but differs in the nonlinear constraints that are analyzed at a given node. Here, we analyze violation of *important* nonlinear constraints

Table 10 (Top) Comparing *qg* and *qgns1* using LS method for different values of τ on TS_1. *qg* could solve 113 instances. (Bottom) Break-up of results over instances of varying difficulty for the best setting $\tau = 2$

τ	# Solved by		Time		Nodes	
	qgns1	Both	*qg*	Rel.	*qg*	Rel.
0.75	115	113	34.71	0.98	7.8e3	1.00
1	114	113	34.71	0.99	7.8e3	1.01
1.5	114	113	34.71	0.95	7.8e3	0.98
2	113	113	34.71	0.93	7.8e3	0.98

Time	# Solved by	Time		Nodes	
	Both	*qg*	Rel.	*qg*	Rel.
> 0	113	34.71	0.93	7.8e3	0.98
> 10	54	191.54	0.90	1.8e5	0.98
> 100	32	535.64	0.88	5.8e5	0.97
> 500	16	1261.25	0.90	1.4e6	0.98

Table 11 (Top) Comparing *qg* and *qgns1* using LS method for different values of τ on TS_2. *qg* could solve 179 instances. (Bottom) Break-up of performance over instances of varying difficulty for the best setting $\tau = 2$

τ	# Solved by		Time		Nodes	
	qgns1	Both	*qg*	Rel.	*qg*	Rel.
0.75	178	177	11.86	0.91	1.0e3	0.89
1	178	177	11.86	0.90	1.0e3	0.90
1.5	178	177	11.86	0.91	1.0e3	0.92
2	178	177	11.86	0.89	1.0e3	0.92

Time	# Solved by	Time		Nodes	
	Both	*qg*	Rel.	*qg*	Rel.
> 0	177	11.86	0.89	1.0e3	0.92
> 10	63	57.76	0.83	1.1e4	0.81
> 100	16	406.47	0.84	1.8e4	0.87
> 500	7	1548.07	0.85	3.8e4	1.03

only. A nonlinear constraint with index k is said to be important based on a surrogate value for its dual multiplier. Let I be the index set of important constraints and is constructed as follows. Given a feasible solution x^l to (R), let d_k be the dual multiplier of the nonlinear constraint with index k at x^l, $d_{max} = \max_{k=1,\ldots,m} d_k$ be the maximum dual value among all the nonlinear constraints, and $\tilde{d}(\leq 1)$ be a positive parameter. We include indices of those nonlinear constraints in set I whose associated dual values are at least $\tilde{d}$ times of the maximum dual value d_{max}. Initially, set I

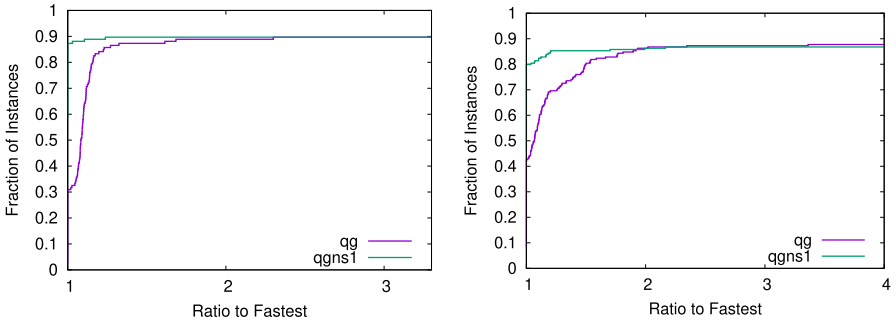

Fig. 7 Performance profiles comparing solution times of *qg* and *qgns1* using LS and $\tau = 2$ on instances in TS_1 (left) and in TS_2 (right)

is populated using x^0, an optimal solution of (R), and is recomputed every time the upper bound is updated using the corresponding solution. Since, the same dual multiplier values are used until a better solution is obtained, we call these values surrogate. For determining points for generating linearizations, the same two methods, ECP and LS, as in *qgns1* are used. The ECP method is slightly modified to consider only important constraints (in set I) for generating linearizations. Also, problems with nonlinearity only in the objective function are treated as in *qgns1*.

In experiments using *qg* with NS2 (*qgns2*), we used the same values for parameter τ, D, and $\tilde{K}$. We used $\tilde{d} = 0.5$ for constructing set I. Again, on both the test sets, LS method for selecting points for linearizations has performed better than ECP. Tables 12 and 13 illustrate results from *qgns2* with LS method on TS_1 and TS_2 respectively. On TS_1, we obtained an improvement of about 7% and of about 13% on TS_2 in solution times. *qgns2* has performed better than *qgns1* on both TS_1 and TS_2.

Table 12 (Top) Comparing *qg* and *qgns2* with LS method and various values of τ on TS_1. *qg* could solve 113 instances. (Bottom) Break-up of results over instances of varying difficulty for best setting $\tau = 0.75$

τ	# Solved by		Time		Nodes	
	qgns2	Both	*qg*	Rel.	*qg*	Rel.
0.75	114	113	34.71	0.93	7.8e3	0.98
1	113	113	34.71	0.93	7.8e3	0.98
1.5	113	113	34.71	0.93	7.8e3	0.98
2	113	113	34.71	0.92	7.8e3	0.98

Time	# Solved by	Time		Nodes	
	Both	*qg*	Rel.	*qg*	Rel.
> 0	113	34.71	0.93	7.8e3	0.98
> 10	54	191.54	0.89	1.8e5	0.98
> 100	32	535.64	0.86	5.8e5	0.97
> 500	16	1261.25	0.92	1.4e6	1.02

Table 13 (Top) Comparing *qg* and *qgns2* with LS method and various values of τ on TS_2. *qg* could solve 179 instances. (Bottom) Break-up of results over instances of varying difficulty for best setting $\tau = 0.75$

τ	# Solved by		Time		Nodes	
	qgns2	Both	*qg*	Rel.	*qg*	Rel.
0.75	178	177	11.86	0.87	1.0e3	0.89
1	178	177	11.86	0.87	1.0e3	0.90
1.5	178	177	11.86	0.89	1.0e3	0.91
2	178	177	11.86	0.88	1.0e3	0.91

Time	# Solved by	Time		Nodes	
	Both	*qg*	Rel.	*qg*	Rel.
> 0	177	11.86	0.87	1.0e3	0.89
> 10	63	57.76	0.81	1.6e4	0.81
> 100	15	463.33	0.82	2.2e4	0.93
> 500	7	1548.07	0.81	3.8e4	1.03

Profiles in Fig. 8 compare the solution times of *qg* and *qgns2* on TS_1 and TS_2. We observe that *qgns2* is faster on about 60% of the total instances in TS_1, and on about 50% of the total instances in TS_2.

3.2.3 Node linearization scheme 3 (NS3)

In this scheme, we use both nonlinear constraints violation and their dual multipliers for deciding whether to select the given node for generating linearizations. First, we compute a score $\hat{s}$ for the node as $\hat{s} = \sum_{k:v_k>0}(V^k + v_k \times d_k)/N$ where V^k and d_k are as defined in schemes NS1 and NS2 and N is the number of violated nonlinear constraints ($v_k > 0$) at x', an optimal solution to the LP relaxation of the node. If the score of the node is more than its parent's score ($\hat{p}$) by at least τ times, then we consider the node for generating linearizations. First, parameter τ is initialized by a preset value. As the tree grows, parameter τ is updated at every selected node (for adding linearizations) by taking its average with $\tilde{\tau} = \hat{s}/(\hat{p} + \epsilon)$, where ϵ is a small tolerance value which in our experiments is 0.001. This scheme is also implemented up to a depth D in the search-tree. Methods for finding linearization points and treatment to problems with nonlinearity only in the objective remain same as in NS1.

In *qg* with NS3 (*qgns3*), we used $\tau = 0.5, 0.75, 1, 1.5$ and the same D as in *qgns1* and *qgns2*. We observed that the ECP method performed better on test set TS_1 and LS performed better on TS_2. In the former case, best performance is obtained using $\tau = 1.5$ where *qgns3* with ECP solved one instance more than *qg* with an improvement of about 11% in the solution time and of about 10% in the number of nodes processed. On TS_2, using $\tau = 0.5$ and LS method, *qgns3* solved two more instances than *qg* with an improvement of about 10% in solution times. These results are presented in Tables 14 and 15. Performance profiles in Fig. 9 present a comparison of the solution times of *qg* and *qgns3* on TS_1 and TS_2. We see that *qgns3* is faster on about

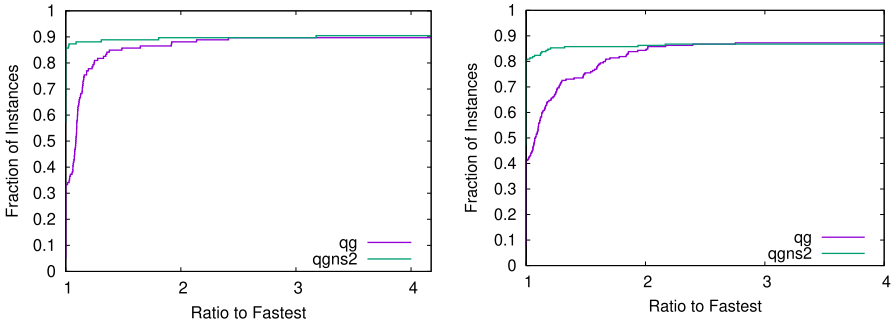

Fig. 8 Performance profiles comparing solution times of *qg* and *qgns2* using LS and $\tau = 0.75$ on instances in TS_1 (left) and in TS_2 (right)

Table 14 (Top) Comparing *qg* and *qgns3* with ECP method and different values of τ on TS_1. *qg* could solve 113 instances. (Bottom) Break-up of results over instances of varying difficulty for best setting $\tau = 1.5$

τ	# Solved by		Time		Nodes	
	qgns3	Both	*qg*	Rel.	*qg*	Rel.
0.5	113	111	31.54	0.94	6.9e3	0.89
0.75	113	110	30.21	0.93	6.9e3	0.89
1	114	111	31.54	0.93	6.9e3	0.90
1.5	114	111	31.54	0.89	6.9e3	0.90

Time	# Solved by	Time		Nodes	
	Both	*qg*	Rel.	*qg*	Rel.
> 0	111	31.54	0.89	6.9e3	0.90
> 10	53	163.83	0.86	1.5e5	0.88
> 100	32	430.23	0.82	4.5e5	0.83
> 500	16	1000.65	0.85	1.1e6	0.84

45% of the total instances in both the test sets. This scheme has performed better than *qgns1* and *qgns2* on both TS_1 and TS_2.

3.2.4 Combination of linearization schemes at root and other nodes

We studied the effects of schemes obtained by combining linearization schemes at the root node with those for fractional nodes. We present a hybrid scheme (Hyb) that automatically identifies structure (S) in a problem and applies linearization schemes RS1 and NS3. For problems (in TS_2) without this structure, Hyb scheme employs RS5 and NS3. Results from *qg* using the hybrid scheme Hyb (*qgHyb*) on TS_1 and TS_2 are detailed in Tables 16 and 17 respectively. These results show that on TS_1, *qgHyb* (with $K = 0.05$ and $\tau = 1.5$) has solved 2 instances more than *qg* with an improvement of about 11% in solution times and about 22% in the number of nodes

Table 15 (Top) Comparing *qg* and *qgns3* with LS method and different values of τ on TS_2. *qg* could solve 179 instances. (Bottom) Break-up of results over instances of varying difficulty for best setting $\tau = 0.5$

τ	# Solved by		Time		Nodes	
	qgns3	Both	*qg*	Rel.	*qg*	Rel.
0.5	181	179	11.72	0.90	1.0e3	0.96
0.75	180	179	11.72	0.90	1.0e3	0.95
1	180	179	11.72	0.92	1.0e3	0.96
1.5	180	179	11.72	0.93	1.0e3	0.95
Time	# Solved by	Time		Nodes		
	Both	*qg*	Rel.	*qg*	Rel.	
> 0	179	11.72	0.90	1.0e3	0.96	
> 10	63	58.02	0.87	1.1e4	0.94	
> 100	15	463.33	0.77	2.2e4	0.91	
> 500	7	1548.07	0.81	3.8e4	1.02	

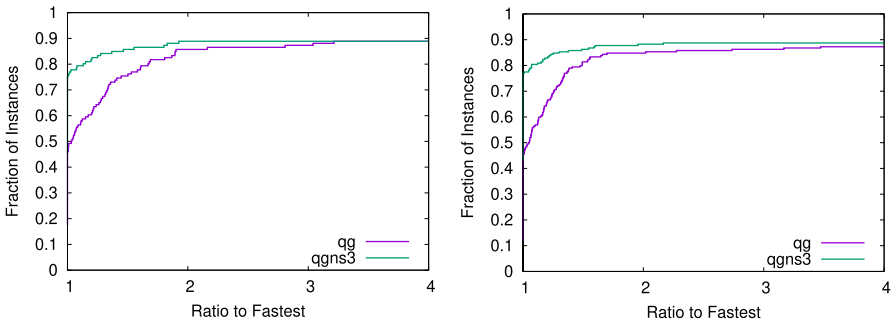

Fig. 9 Performance profiles comparing solution times of *qg* and *qgns3* using ECP and $\tau = 1.5$ on instances in TS_1 (left) and using LS and $\tau = 0.5$ on instances in TS_2 (right)

Table 16 (Top) Comparison of *qg* and *qgHyb* on TS_1. *qg* could solve 113 instances. (Bottom) Break-up of performance of *qgHyb* over instances of varying difficulty

# Solved by		Time		Nodes	
qgHyb	Both	*qg*	Rel.	*qg*	Rel.
115	112	33.01	0.89	7.3e3	0.78
Time	# Solved by	Time		Nodes	
	Both	*qg*	Rel.	*qg*	Rel.
> 0	112	33.01	0.89	7.3e3	0.78
> 10	54	172.97	0.86	1.6e5	0.82
> 100	34	430.69	0.81	4.7e5	0.75
> 500	16	1115.30	0.89	1.2e6	0.80

Table 17 (Top) Comparison of *qg* and *qgHyb* on TS_2. *qg* could solve 179 instances. (Bottom) Break-up of performance of *qgHyb* over instances of varying difficulty

# Solved by		Time		Nodes	
qgHyb	Both	*qg*	Rel.	*qg*	Rel.
180	177	11.86	0.87	1.0e3	0.86
Time	# Solved by	Time		Nodes	
	Both	*qg*	Rel.	*qg*	Rel.
> 0	177	11.86	0.87	1.0e3	0.86
> 10	64	56.37	0.81	1.1e4	0.84
> 100	16	408.04	0.82	2.1e4	0.85
> 500	7	1548.07	0.69	3.8e4	0.85

processed. On TS_2, *qgHyb* (with $\theta = 2$ and $\tau = 1.5$) has solved one instance more than *qg* with about 13% reduction in both solution times and nodes processed. Overall, on test set *TS*, we solved 3 more instances and obtained an improvement of about 12% in the solution times over *qg*. A comparison of solution times of *qg* and *qgHyb* on test sets TS_1 and TS_2 is reported in Fig. 10. *qgHyb* is faster on about 40% of the total instances in both the test sets. Furthermore, *qgHyb* is two times faster on about 15% of the total instances in TS_1.

Profiles in Fig. 11 show a consolidated comparison of the solution times of the above schemes and the default *qg* algorithm for test sets TS_1 and TS_2. We observe that *qgHyb* performs better than the others on both the test sets. On about 80% of the total instances (in each test set), the solution times of *qgHyb* are within 1.5 times the fastest solver.

4 Shared-memory parallel search

We deploy a parallel tree-search algorithm for solving different nodes of the branch-and-bound tree concurrently using different processors that share a common memory. All open subproblems (associated with nodes) of the branch-and-bound tree are stored in a collection called the node-pool. Different nodes are solved in parallel using the *fork-join* model, a commonly used multiprocessing model in shared-memory architectures. The main program is run as a single process which creates multiple 'threads' [17, 26] depending on the number of CPUs available and user settings. Threads are capable of doing mutually independent computations like processing different nodes concurrently.

The fork-join model can be thought of as an alternating sequence of forks where various tasks are performed concurrently by multiple threads, and joins, where a single thread performs some serial tasks and synchronization for sharing information between the threads. In our implementation, the main process first reads the MINLP instance, performs some preprocessing and sets up the environment and

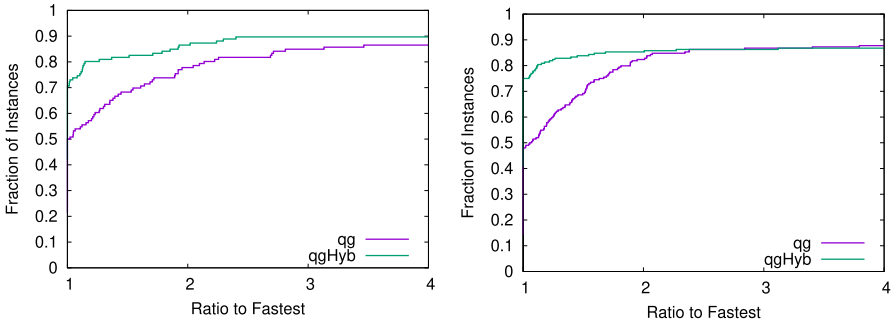

Fig. 10 Performance profiles comparing solution times of *qg* and *qgHyb* on instances in TS_1 (left) and in TS_2 (right)

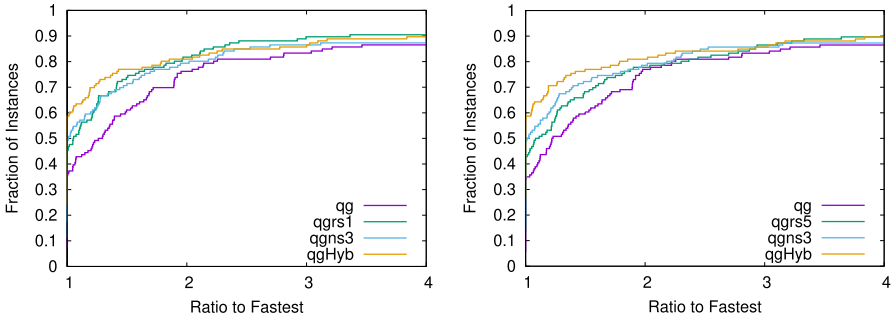

Fig. 11 Performance profiles comparing solution times of solvers on instances in TS_1 (left) and in TS_2 (right)

other required data structures. The main process also creates the threads and starts branch-and-bound. Branch-and-bound then proceeds in rounds. Every thread selects an open-node and removes it from the node-pool. Only one thread is allowed to access the node-pool at a time and other threads wait for their turn. If there are no nodes available for a thread, it waits until the next round. Once this selection process is completed, all threads concurrently start solving their respective nodes. When all the threads finish solving their respective nodes, a new round of assignment of open-nodes and solving is executed. This process continues until all the open-nodes are either processed or pruned and the node-pool becomes empty. We use this fork-join node-level parallelism for two algorithms: NLP-BB and QG.

We have implemented our fork-join model using the OpenMP directives [19]. OpenMP directives provide a simple way of specifying concurrency, synchronization and data handling - without the need to explicitly create threads, allocate memory, delete memory etc. While this approach provides fewer features and lesser flexibility than POSES threads (popularly called Pthreads) or standard threads provided by C++11, it simplifies multithreaded programming to a great extent.

While a more detailed description of Minotaur design and its C++ classes is available in [39], we briefly describe the important C++ classes that we use for

our parallel implementation. The program starts by reading the problem, and then *presolves* it using the `Presolver` class. The presolved problem is then passed to the `NodeRelaxer` class which creates a relaxation. A node is processed using the `NodeProcessor` class, that deploys an appropriate LP or NLP solver called through the `LPEngine` or `NLPEngine` class. If an optimal solution of the relaxation is found, and if this solution is not feasible to the MINLP, the `NodeProcessor` calls a `Brancher` class to find a suitable branching candidate. The class `TreeManager` handles all the tree-related information: nodes, upper and lower bounds, etc. Using the branches found by the `Brancher`, two new child nodes are created by the `TreeManager`.

We preserve the basic design of the sequential branch-and-bound in Minotaur and utilize the existing classes, which makes our implementation light-weight. As in the serial version, we maintain a single, central `TreeManager` which stores and maintains all node descriptions. Each thread individually maintains a private copy of all the necessary class objects, like `NodeRelaxer`, `NodeProcessor`, `Brancher` etc., and acts as an independent unit, that synchronizes with other threads at the end of each round. The first thread starts solving the root relaxation while the other threads wait. If branching is required, the thread creates two child nodes. In the next round of node selection, one of the other idle threads obtains a node. Each thread that has a node now processes its respective node in the next round and the process continues. When sufficient number of open-nodes are available, all threads become busy. If T number of threads are used, the ramp-up time before all threads are busy is at least $\lceil log_2(T) \rceil$ times the average node solving time. When the node-selection strategy is based on diving [20], each thread retains one of the children of the node it solves for quick warm-starting of LPs or NLPs. Each thread maintains a private copy of the original MINLP to create relaxations of the nodes that it receives and to check whether a relaxation yields a feasible solution to the MINLP. After each round of solving, stopping conditions are checked by any one of the threads. The search terminates when all open-nodes are exhausted (solved, pruned by bound or pruned by infeasibility) or some other stopping condition (time limit, node limit etc.) is met. The schematics of the parallel tree-search and the *Process* block are shown in Fig. 12.

4.1 Parallel extension of NLP-BB

The scheme shown in Fig. 12 can be viewed as the parallel NLP-BB algorithm, where the nodes in the tree are NLP relaxations and an `NLPEngine` (NLP subroutine) is used to solve them. We denote this parallel solver in Minotaur as *mcbnb* and study its performance when using different number of threads. The hardware and software setup mentioned in Sect. 3 has been used in these experiments as well. The NLP solver IPOPT [56] (version 3.12) with MA97 linear-systems solver is thread-safe, hence suitable for our parallel algorithm.

The scalability of our implementation with the number of threads is depicted by what we call a 'Scalability Graph'. While SGM gives the mean improvement over all instances, this graph shows the distribution of performance over the test set. It is

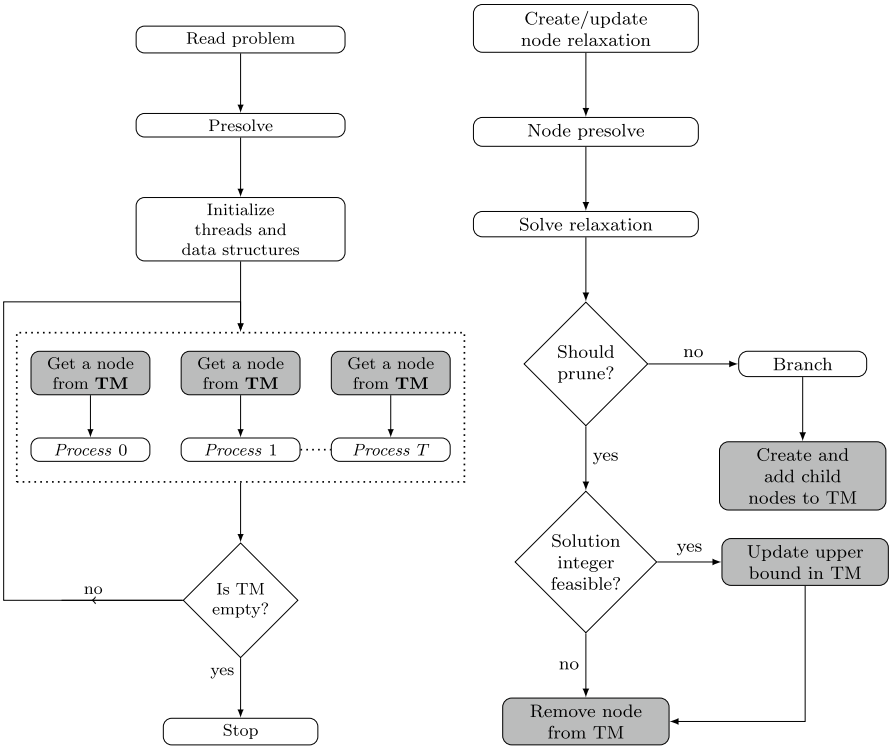

Fig. 12 Schematics of the parallel tree-search (left) in Minotaur and the *Process* block (right). Gray-colored blocks involving the `TreeManager` (denoted **TM**) are critical. The block where stopping condition is checked is executed by any one of the threads

a line plot with each line corresponding to a fixed thread-count. Each line plots the fraction of instances that can be solved within a w-factor of time taken by the single-thread run. Given a set of instances, TS, the graph is plotted as a non-decreasing line graph. For each value w, it plots

$$\frac{\left|\{i \in TS : t_{i,T} \leq w t_{i,1}\}\right|}{|TS|},$$

where $t_{i,T}$ is the time taken by the solver when running T threads on instance i. If the solver does not finish solving within the time limit, $t_{i,T}$ is set to infinity. The ratios we use are different from the ones used in performance profiles [21], where the ratios are calculated with respect to the time taken by the fastest solver for each instance.

Figure 13 (left) shows the scalability graphs for *mcbnb*. The plot for *mcbnb1* (*mcbnb* with one thread), the reference solver, is a step function by definition. Its height (about 0.7 in this case) is the fraction of instances that could be solved within the time limit by the single-thread run. The plot for *mcbnb2* shows that it could solve about 5% (value at 2^{-1}) of the instances faster by a factor of two or more as

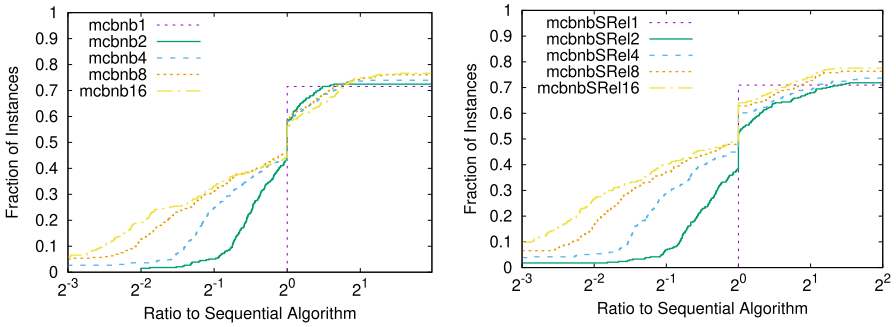

Fig. 13 Scalability graphs of wall clock times taken by multithreaded variants of *mcbnb* (left) and *mcbnbSRel* (right) on test set *TS*

compared to *mcbnb1*. Similarly, *mcbnb4* and *mcbnb16* could solve about 20% and 30% respectively for the same. The rightmost value on the plot shows the fraction of instances that could be solved within the time limit.

SGM values for wall clock time and nodes processed are reported in Table 18 along the lines of tables in Sect. 3. The first column ('# Threads (T)') in the top table in Table 18 indicates the number of threads used. A 'T' at the end of the solver name indicates the number of threads used by it. Also, 'Wall time' denotes the wall clock time (not the CPU time) taken by the multithreaded code. Using 16 threads, *mcbnb* could solve 17 additional instances compared to *mcbnb1*, and achieved a speed-up of about two on average. The growth in tree-size with increasing number of threads is well below linear, which ultimately leads to gains in parallelism. The bottom table in Table 18 shows the statistics for *mcbnb16* when instances are categorized based on difficulty level. The improvements due to parallelism are more prominent for difficult instances (row corresponding to time > 100).

4.2 Sharing pseudocosts in branching

The implementation of NLP-BB and QG algorithms in Minotaur use the well-known reliability branching scheme [6]. Reliability branching uses strong branching [7, 36] initially to find the score of branching candidates. As strong branching is expensive, the scheme uses previously computed scores after a certain number of strong-branching trials. In a parallel setting, the scores obtained at a node by a thread may be useful at nodes processed by other threads. However, sharing this information comes at the cost of querying additional information (from other threads), which means that each thread has to spend additional time in gathering and processing this information.

We implemented reliability branching for a parallel setting in two different ways. In the first way which we call *privateRel*, each thread does reliability branching independent of other threads using information from only the nodes that it processed earlier. In the second way which is referred to as *sharedRel*, each thread uses information from the nodes solved by other threads also. This

Table 18 (Top) Comparison of *mcbnb1* with *mcbnb* using multiple threads on test set *TS*. *mcbnb1* could solve 239 instances. (Bottom) Break-up of performance of *mcbnb16* over instances of varying difficulty

# Threads	# Solved by		Wall time		Nodes	
(T)	*mcbnbT*	Both	*mcbnb1*	Rel.	*mcbnb1*	Rel.
2	242	238	32.73	0.80	3.1e2	1.05
4	247	239	33.53	0.69	3.2e2	1.16
8	254	239	33.53	0.60	3.2e2	1.28
16	256	239	33.53	0.56	3.2e2	1.54

Time	# Solved by	Wall time		Nodes	
	Both	*mcbnb1*	Rel.	*mcbnb1*	Rel.
> 0	239	33.53	0.56	3.2e2	1.54
> 10	132	113.58	0.41	9.6e2	1.57
> 100	63	428.05	0.28	2.6e3	1.40
> 500	25	1153.12	0.30	5.3e3	1.60

aspect is illustrated in Fig. 14. Suppose, for instance, we have two threads, then the first thread, `thread0`, solves the root node indexed 0 in the first round and then one gray-colored node in each subsequent round. Simultaneously, the other thread, `thread1`, starts solving the hatched nodes, starting from the the node indexed 2. In *privateRel*, both `thread0` and `thread1` use the information generated only at the nodes they solve. The other brancher, *sharedRel*, queries the node-solve information from the other threads at the end of each round and uses the cumulative information (from both gray-colored and hatched nodes) to decide the branching variable at a node. The accumulation of information like pseudocosts [6] etc. from other threads to calculate scores requires additional memory-reads and some computations at each thread.

Figure 13 (right) shows the effect of sharing pseudocosts in *mcbnb* (*mcbnbSRel*) when using multiple threads. We see that sharing pseudocosts after each round is beneficial, and the benefits grow with the number of threads. As shown in Table 19, sharing pseudocosts enabled *mcbnbSRel16* to solve 4 more instances than *mcbnb16* (21 more compared to *mcbnb1*). Also, the mean wall clock time is reduced to a fourth for difficult instances (row corresponding to time > 500) using *mcbnbSRel16*.

4.3 Parallel extension of the QG algorithm

The implementation of parallel QG algorithm in Minotaur differs from *mcbnb* in two ways. First, an LP solver is used to solve the (LP) relaxations at each node. Second is the generation and sharing of globally valid linearization cuts that are generated at certain nodes either after solving an NLP or by linearization methods like those described in Sect. 3.

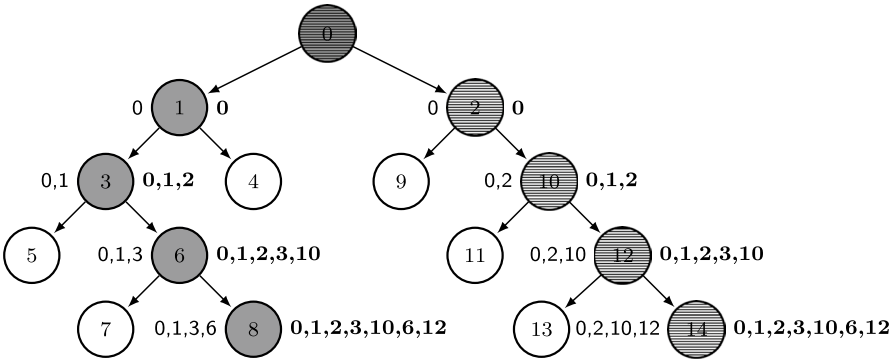

Fig. 14 Illustration of using pseudocosts by two threads for branching. The root node indexed 0 and then the gray-colored nodes are solved by `thread0` and the hatched nodes (except 0) are solved by `thread1`. In *privateRel*, `thread0` uses pseudocosts only from the gray-colored nodes while `thread1` uses pseudocosts from only the hatched nodes (indices shown on the left of each node). In *sharedRel*, information from all the processed nodes is used by both the threads (indices shown on the right of each node)

Table 19 (Top) Comparison of *mcbnbSRel1* with *mcbnbSRel* using multiple threads on test set *TS*. *mcbnbSRel1* could solve 237 instances. (Bottom) Break-up of performance of *mcbnbSRel16* over instances of varying difficulty

# Threads	# Solved by		Wall time		Nodes	
(T)	*mcbnb-SRelT*	Both	*mcbnb-SRel1*	Rel.	*mcbnb-SRel1*	Rel.
2	241	236	30.71	0.86	3.0e2	1.11
4	247	236	30.73	0.69	3.0e2	1.24
8	255	237	30.23	0.56	3.1e2	1.40
16	260	237	30.23	0.50	3.1e2	1.59

Time	# Solved by	Wall time		Nodes	
	Both	*mcbnbSRel1*	Rel.	*mcbnbSRel1*	Rel.
> 0	237	31.23	0.50	3.1e2	1.59
> 10	130	104.24	0.37	9.0e2	1.64
> 100	63	364.46	0.27	2.1e3	1.61
> 500	23	1008.44	0.25	4.8e3	1.65

In order to store and share these cuts, first we add them to a local `CutPool` of the respective thread. A `CutManager` class is used by each individual thread to store all the linearizations generated while processing the nodes assigned to it. A thread queries the `CutManager` of all other threads while creating the relaxation of the respective node, and all cuts that are *new* for this thread are added to this relaxation. The cuts from `CutManagers` of different threads that have been added to the relaxation at a given

thread are maintained and updated using a unique cut id. We denote this parallel QG algorithm as *mcqg*. Algorithm 4 demonstrates the *mcqg* algorithm implemented within Minotaur, and Algorithm 5 describes the function *GetNode*() used in Algorithm 4.

Algorithm 4: Parallel QG (LP/NLP based branch-and-bound) algorithm in Minotaur.

1 Initialize upper bound, $U = \infty$, state of thread, $S_t = idle$, cut pool, $\mathcal{C}_t = \emptyset$, $\forall\, t \in 1 \ldots T$.
2 Add root LP relaxation to the pool of open-nodes $\mathcal{H}$.
3 **while** $\mathcal{H} \neq \emptyset$ **do**
4 **for** t *in* $1 \ldots T$ **do**
5 **if** $S_t = idle$ **then**
6 *GetNode()*.
7 **if** $S_t = assigned$ **then**
8 Add *new* cuts from $\mathcal{C}_t$, $\forall t$ in $1 \ldots T$, to LP_t.
9 Solve LP_t at thread t.
10 **if** LP_t *is optimal and* $(\hat{x}^t)_i \in \mathbb{Z}$, $\forall\, i \in \mathcal{I}$ **then**
11 Solve F-NLP($\hat{x}^t$), let the point returned be $\check{x}^t$.
12 **if** *F-NLP*($\hat{x}^t$) *is optimal* **then**
13 Update $U \leftarrow \min\{U, f(\check{x}^t)\}$.
14 Generate linearizations of all nonlinear constraints violated by $\hat{x}^t$, at $\check{x}^t$, and add to $\mathcal{C}_t$ and LP_t.
15 Go to step 9.
16 **else if** LP_t *is infeasible* **then**
17 Prune this node.
18 *GetNode()*.
19 **else**
20 Branch: generate two LP subproblems and add to $\mathcal{H}$.
21 *GetNode()*.

Algorithm 5: Get an open-node from $\mathcal{H}$ for a thread $t \in \{1, \ldots, T\}$

1 **if** $\mathcal{H} \neq \emptyset$ **then**
2 Remove an LP from $\mathcal{H}$ as per the search strategy and set $LP_t \leftarrow LP$ and $S_t = assigned$.
3 **else**
4 Set $S_t = idle$.

Table 20 summarizes the performance of multithreaded variants of *mcqg* relative to *mcqg1*. All threads share linearizations (at integer solutions) and pseudocosts according to *sharedRel* scheme. We observed improvements with all the variants of *mcqg* over *mcqg1*. About 44% improvement in wall clock time is obtained when using 16 threads and 9 more instances were solved. The scalability graphs for *mcqg* are shown in Fig. 15.

Table 20 (Top) Comparison of *mcqg1* to *mcqg* using multiple threads on test set *TS*. *mcqg1* could solve 285 instances. (Bottom) Break-up of performance of *mcqg16* over instances of varying difficulty

# Threads	# Solved by		Wall time		Nodes	
(T)	*mcqgT*	Both	*mcqg1*	Rel.	*mcqg1*	Rel.
2	283	279	19.44	0.86	2.1e3	1.08
4	291	282	19.50	0.75	2.1e3	1.17
8	291	280	19.62	0.64	2.1e3	1.20
16	294	284	20.08	0.56	2.2e3	1.29

Time	# Solved by	Wall time		Nodes	
	Both	*mcqg1*	Rel.	*mcqg1*	Rel.
> 0	284	20.08	0.56	2.2e3	1.29
> 10	120	100.52	0.39	3.3e4	1.32
> 100	56	331.22	0.27	1.2e5	1.30
> 500	19	1199.84	0.24	4.1e5	1.19

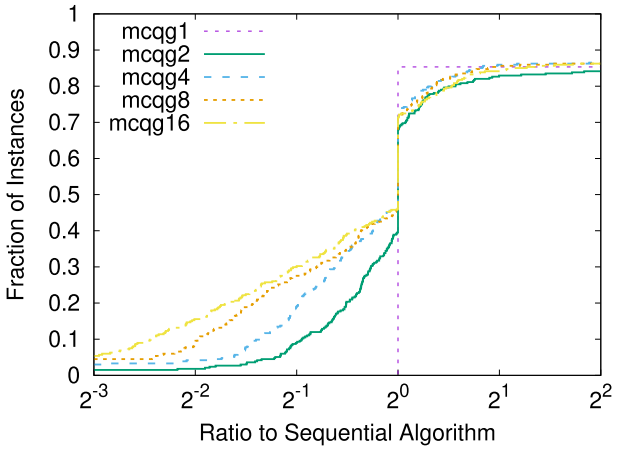

Fig. 15 Scalability graphs of wall clock times for *mcqg* variants on test set *TS*

5 Combined effect of linearization and parallelization schemes

Our numerical experiments show that deploying linearization schemes within parallel tree-search further enhances the performance of *qg* and *mcqg* algorithms. We show the performance of *mcqg* with the hybrid linearization scheme Hyb presented in Sect. 3.2.3. We refer to the combination of *mcqg* with Hyb as *mcqgHyb* and compare it to *qg* and *mcqg16*. Tables 21 and 22 show the performance of *mcqgHyb16* (*mcqgHyb* with 16 threads) on TS_1 and TS_2, respectively. Note that the wall clock time taken by the sequential algorithm *qg* is the same as the CPU time. Using *mcqgHyb16* on TS_1, we observed a significant improvement of about 52% in

Table 21 (Top) Comparison of *mcqg16* and *mcqgHyb16* to *qg* on test set TS_1. *qg* could solve 113 instances. (Bottom) Break-up of results of *mcqgHyb16* over instances of varying difficulty

Method	# Solved by		Wall time		Nodes	
(M)	M	Both	*qg*	Rel.	*qg*	Rel.
mcqg16	115	111	31.54	0.54	6.9e3	1.35
mcqgHyb16	119	112	33.01	0.48	7.3e3	1.07

Time	# Solved by	Wall time		Nodes	
	Both	*qg*	Rel.	*qg*	Rel.
> 0	112	33.01	0.48	7.3e3	1.07
> 10	53	181.01	0.32	1.7e5	1.09
> 100	31	504.06	0.27	5.3e5	0.94
> 500	16	1021.32	0.31	1.1e6	1.19

Table 22 (Top) Comparison of *qg* and *mcqgHyb16* on test set TS_2. *qg* could solve 179 instances. (Bottom) Break-up of results of *mcqgHyb* over instances of varying difficulty

Method	# Solved by		Wall time		Nodes	
(M)	M	Both	*qg*	Rel.	*qg*	Rel.
mcqg16	179	177	11.86	0.88	1.0e3	1.45
mcqgHyb16	181	176	11.42	0.62	1.0e3	1.19

Time	# Solved by	Wall time		Nodes	
	Both	*qg*	Rel.	*qg*	Rel.
> 0	176	11.42	0.62	1.0e3	1.19
> 10	64	52.15	0.47	1.1e4	1.08
> 100	14	445.50	0.35	2.4e4	0.86
> 500	6	1727.45	0.28	5.2e4	0.81

the solution times and solved 6 instances more than *qg*. On TS_2, we solved 2 more instances and obtained an improvement of about 38% in the solution times.

6 Outer-approximation with parallelism in MILP solving

As briefly explained in Sect. 1, the underlying strategy in outer-approximation based algorithms is to solve an alternating sequence of MILP relaxations (RM) and fixed-NLPs (F-NLPs). In this section, we describe two versions of OA implemented in Minotaur where we exploit parallelism of the MILP solver.

6.1 Multitree OA with parallel MILP solving

As OA is an iterative scheme in which an MILP and a fixed-NLP are solved alternatingly, a natural way of parallelizing it is to use a parallel MILP solver. We have implemented the default OA scheme in Minotaur and also enhanced it in the following way. We solve MILP relaxation at any iteration using an MILP solver. The MILP solver can utilize all the available processors. MILP solvers also have the capability of returning a pool of solutions which we use to generate additional linearizations. For each solution x^t in the pool returned by the MILP solver, we solve the corresponding fixed-NLP F-NLP(x^t) and generate the linearizations. These NLPs can in turn be solved in parallel if the NLP solver is thread-safe. All linearizations that are active at the NLP solution are added to the MILP. When all NLPs have been solved and linearizations added, the MILP solver is called again and the process continues. Algorithm 6 describes the steps of the enhanced OA. We also solve multiple F-NLPs, each corresponding to a distinct MILP solution, in parallel (the for loop in Algorithm 6).

Algorithm 6: Exploiting solution pool of MILP solver in multitree OA

1 Initialize bounds, $U = \infty, L = -\infty$, iteration counter $k = 0$.
2 Solve the NLP relaxation (R). If (R) is infeasible, then so is (P) and we STOP. If the optimal solution of (R), x^0, is feasible for (P), then set $U = f(x^0) = L$ and STOP.
3 Create and solve the MILP relaxation (RM). If (RM) is infeasible, then so is (P) and we STOP. Otherwise, let $\hat{X}^k$ be the set of available feasible solutions of (RM), $\hat{z}^k$ be its optimal value, and set $L = \hat{z}^k$.
4 **while** $U > L$ **do**
5 **for** $x^t \in \hat{X}^k$ **do**
6 Solve F-NLP(x^t), let the point returned be $\check{x}^t$. If F-NLP(x^t) is optimal, update $U \leftarrow \min\{U, f(\check{x}^t)\}$.
7 Add linearizations to nonlinear constraints violated by x^t, at $\check{x}^t$, to (RM).
8 Set $k \leftarrow k + 1$, solve (RM), and update $L \leftarrow \hat{z}^k$.

In order to further accelerate the MILP solver, we use the MIP starts functionality provided by the MILP solver, CPLEX in our case. The solutions obtained by it are written to a file and are read in the subsequent MILP call. In our experiments, we observed that CPLEX was able to repair some of the solutions from the MIP starts and obtain upper bounds, mainly because the MILPs in consecutive iterations differ only by a few linear constraints. Additionally, we provide the best known upper bound of (P) to the MILP solver in each iteration to be used as a cut-off value. In Minotaur, we interact with the CPLEX solver using a C++ wrapper that passes information to and from CPLEX through its C interface.

We compare the performance of our two implementations of multitree OA. In the first implementation, linearizations are added only at the point obtained from the optimal solution of MILP. The second one uses all solutions of the solution pool of MILP, and solves fixed-NLPs in parallel using multiple threads. We denote these implementations of OA as *oa* and *oaSol* respectively. Figure 16 shows the scalability

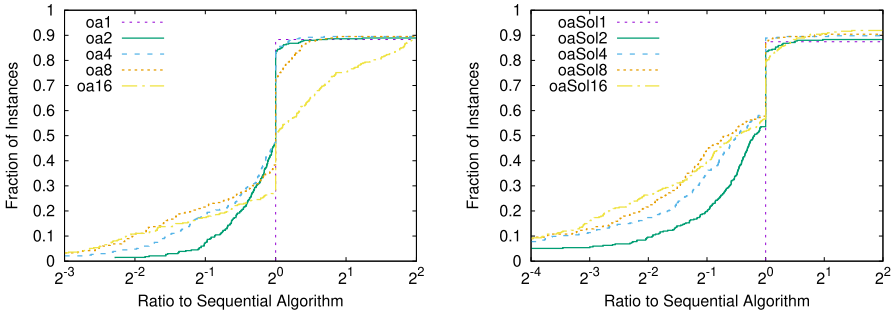

Fig. 16 (Left) Effect of providing multiple threads to CPLEX in *oa* on test set *TS*. (Right) Performance of *oaSol* that uses the solution pool of CPLEX and solves fixed-NLPs in parallel

Table 23 (Top) Comparison of *oa* using multiple threads. *oa1* could solve 296 instances. (Bottom) Break-up of *oa16* results over instances of varying difficulty

# Threads	# Solved by		Wall time		Iterations	
(T)	*oaT*	Both	*oa1*	Rel.	*oa1*	Rel.
2	299	295	11.44	0.80	12.44	1.00
4	301	295	11.44	0.68	12.44	1.01
8	302	295	11.44	0.62	12.44	1.01
16	302	295	11.44	0.84	12.44	1.01

Time	# Solved by	Wall time		Iterations	
	Both	*oa1*	Rel.	*oa1*	Rel.
> 0	295	11.44	0.84	12.44	1.01
> 10	109	54.18	0.67	37.47	1.02
> 100	36	335.70	0.31	48.23	0.98
> 500	12	1161.57	0.22	73.49	0.94

graphs, and Tables 23 and 24 provide a summary of performance of these algorithms. Here, we present the SGM for the number of iterations taken by *oa*. We observe that the use of solution pool enables us to solve more instances. One can also solve fixed-NLPs one-by-one if a thread-safe NLP solver is not available. In our experiments, we found that using the solution pool and solving fixed-NLPs in parallel is the most effective strategy. Compared to the traditional OA (*oa1*), we solved up to 13 more instances and improved the wall clock time by more than 50%.

6.2 QG using MILP solvers with lazy cuts callback

This version of QG is also known as the *Single-tree* OA because it explores a single tree, but uses an MILP solver for creating the tree. MILP solvers like CPLEX and GUROBI provide the users with callback functions which can be invoked

Table 24 (Top) Comparison of *oaSol* using multiple threads. *oaSol1* could solve 290 instances. (Bottom) Results of *oaSol16* over instances of varying difficulty

# Threads	# Solved by		Wall time		Iterations	
(T)	*oaSolT*	Both	*oaSol1*	Rel.	*oaSol1*	Rel.
2	297	288	13.92	0.63	16.75	0.66
4	302	288	13.81	0.50	16.79	0.54
8	304	290	14.17	0.45	16.52	0.60
16	309	290	13.94	0.43	16.69	0.58

Time	# Solved by	Wall time		Iterations	
	Both	*oaSol1*	Rel.	*oaSol1*	Rel.
> 0	290	13.94	0.43	16.69	0.58
> 10	108	67.46	0.27	40.24	0.51
> 100	35	401.20	0.16	62.52	0.58
> 500	14	1255.97	0.09	98.26	0.36

in specific *contexts*, for example, when an integer feasible solution is found in the MILP tree. In such contexts, the MILP solving is paused and the control is transferred (temporarily) to a predeclared user-callback function. The user can access MILP solving information, for example, the best solution, upper and lower bounds etc., generated within the MILP solver so far. This information can then be utilized in the callback to generate new cuts, feasible solutions etc. that are passed back to the MILP solver through predefined functions. When solving convex MINLPs, the MILP solver is not aware of the nonlinear constraints. When an integer feasible solution to the MILP is obtained, it has to be checked for nonlinear constraints. If the solution violates any of them, linearization cuts generated using this point are added to the MILP as 'lazy' cuts, which cut this solution off. In this way, the MILP tree is guided towards an optimal solution of (P). In this algorithm, the MILP solver maintains the MILP tree, along with most of its advanced MILP solving features like presolving, implications, heuristics etc. that help accelerate the overall tree-search.

This implementation is similar to the multitree OA. First, the root MILP relaxation is passed to the MILP solver. Before solving the MILP, we activate the lazy constraints callback function in the MILP solver. Whenever the MILP solver finds an integer feasible solution, say x^t, it returns the control back to Minotaur through a predefined callback. We solve F-NLP(x^t) in the callback, generate linearization cuts for all nonlinear constraints active at the solution and pass them to the MILP solver which then resumes the MILP tree-search. All the available processors are utilized by the MILP solver within its algorithm. We observed that CPLEX sets the parallel tree-search mode to *deterministic* when using the lazy cuts callback, and only one thread is allowed to access the callback at a time. We conducted two sets of experiments: one with the *deterministic* mode and the other by explicitly

Table 25 (Top) Comparison of *lstoaD* using multiple threads. *lstoaD1* could solve 307 instances. (Bottom) Break-up of results of *lstoaD16* over instances of varying difficulty

# Threads	# Solved by		Wall time		Nodes	
(T)	*lstoaDT*	Both	*lstoaD1*	Rel.	*lstoaD1*	Rel.
2	308	306	9.86	0.93	8.6e2	1.05
4	309	306	9.86	0.83	8.6e2	1.07
8	309	306	9.86	0.80	8.6e2	1.09
16	309	306	9.86	0.92	8.6e2	1.17

Time	# Solved by	Wall time		Nodes	
	Both	*lstoaD1*	Rel.	*lstoaD1*	Rel.
> 0	306	9.86	0.92	8.6e2	1.17
> 10	111	42.74	0.79	1.2e4	1.23
> 100	32	299.94	0.41	6.7e4	1.01
> 500	13	871.90	0.25	1.4e5	0.94

setting the parallel mode of CPLEX to *opportunistic*. The latter mode does not guarantee reproducibility of results, so we performed 5 replications. For each instance in test set *TS*, its solution time is computed as the arithmetic mean of the 5 replications. Tables 25, 26 and Fig. 17 present the performance of `deterministic` (*lstoaD*) and `opportunistic` (*lstoaO*) modes. We observed good scalability with *lstoaO*. Using 16 threads, both solution time and tree-size were improved by more than 60%. On the other hand, *lstoaD* did not show scalability,

Table 26 (Top) Comparison of *lstoaO* using multiple threads. *lstoaO1* could solve 307 instances. (Bottom) Break-up of results of *lstoaO16* over instances of varying difficulty

# Threads	# Solved by		Wall time		Nodes	
(T)	*lstoaOT*	Both	*lstoaO1*	Rel.	*lstoaO1*	Rel.
2	317	307	12.32	0.74	8.6e2	0.07
4	318	305	12.35	0.57	8.6e2	0.06
8	323	307	12.32	0.44	8.6e2	0.08
16	325	307	12.32	0.37	8.6e2	0.07

Time	# Solved by	Wall time		Nodes	
	Both	*lstoaO1*	Rel.	*lstoaO1*	Rel.
> 0	307	12.32	0.37	8.6e2	0.07
> 10	109	57.27	0.22	1.3e4	0.01
> 100	28	453.24	0.10	9.3e4	0.00
> 500	13	1024.61	0.08	8.8e4	0.01

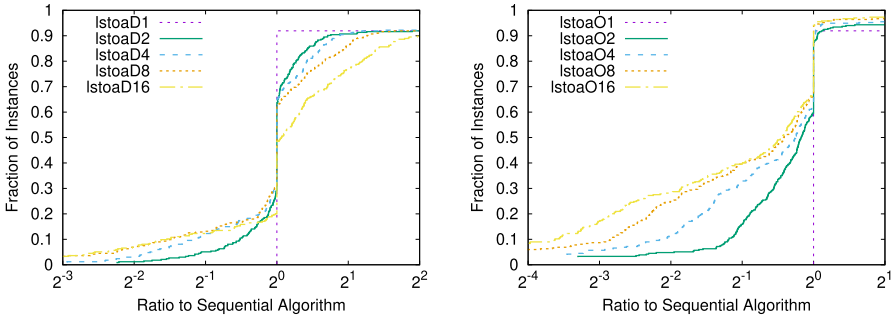

Fig. 17 Effect of providing multiple threads to *lstoaD* and *lstoaO* (*qg* implemented using CPLEX with lazy cuts callback functionality using *deterministic* (left) and *opportunistic* (right) parallel mode) on test set *TS*

probably due to the sequential NLP solving. Although, both *qg* and *lstoa* are implementations of QG, use of advanced MILP solving techniques within *lstoa* leads to better performance when compared to *qg*. We discuss it next.

7 Comparison of methods

In the next part of our study, we compare these enhanced routines to each other and also to other MINLP solvers. The goal of this comparison is not to benchmark these solvers, but rather to understand the broad effects of the choice of algorithms and implementation details on the performance. We consider the serial and parallel versions of four algorithms described in this paper: NLP-BB with sharing of branching information between threads (*mcbnbSRel*), QG with extra linearizations and parallelization using our own branch-and-cut implementation (*mcqgHyb*), QG with branch-and-cut implementation of CPLEX MILP solver running in opportunistic mode (*lstoaO*), and OA with CPLEX MILP solver using all solutions from CPLEX's solution pool (*oaSol*).

We also include two other MINLP solvers that support parallelization: FSCIP [53] and SHOT [37]. FSCIP is a shared-memory variant of the MILP and MINLP solver SCIP [4]. SCIP was initially developed for MILP and was later extended [55] to global optimization. Developed in C language, it has several plugins that exploit problem structure for branching, presolving, heuristic search, cutting planes, conflict analysis etc. SCIP can call several LP solvers including CPLEX and also the NLP solver IPOPT for solving relaxations. As mentioned in Sect. 2.2, FSCIP uses the UG framework to call separate SCIP instances at each thread. Open subproblems are distributed to each thread which then solve the respective subtrees. UG also dynamically controls and manages the load at each thread. SHOT was developed recently for solving convex MINLPs. It implements ESH and ECP based algorithms, similar to outer-approximation, that solve a sequence of MILP subproblems. SHOT also has a lazy cuts based QG algorithm. SHOT depends on parallelism that the MILP solver

Table 27 Comparison of algorithms deployed by different solvers along with the SGM of wall clock times and number of instances solved from set *TS*

				Single thread		16 Threads	
Solver	Algorithm	Relaxation	Branch-and-cut\ bound implementation	# Solved	Wall time	# Solved	Wall time
mcbnbSRel	NLP-BB	NLP	own	237	31.23	260	25.24
fscip	QG	LP	own	276	14.99	273	5.93
shot	QG	LP	MILP solver	309	8.75	309	6.40
mcqgHyb	QG	LP	own	295	17.52	300	12.66
lstoaO	QG	LP	MILP solver	307	12.32	325	6.66
oaSol	OA	MILP	MILP solver	295	11.63	309	8.19

exploits in both these algorithms. For our experiments, we use the default ESH and lazy cuts based QG algorithm (also called single-tree polyhedral outer-approximation by SHOT [37]).

We compiled SCIP, SHOT and Minotaur using the same versions of CPLEX (LP and MILP) and IPOPT (NLP) subsolvers. Also, we maintained all the default settings of these solvers except in FSCIP, where we disabled convexity detection routines by setting `constraints/nonlinear/assumeconvex` to `True`. Table 27 summarizes the key differences in the basic algorithms, implementation of branch-and-cut routines and the performance on the test set *TS*. Unlike earlier tables, the SGM of the wall clock times is computed over the instances solved by the particular solver and does not depend on any other solver. We see that all solvers benefit from parallelization, although without good scalability. We also see that OA with a state-of-the-art branch-and-cut MILP solver performs better than the QG algorithm with one's own branch-and-cut implementation that may lack several key MILP features. Implementing QG using callbacks to a fast commercial MILP solver seems to be the best option. This option is however encumbered by the availability and licensing of the MILP solver. QG with enhanced linearization schemes with one's own branch-and-cut is seen to be the next best option.

8 Some large scale experiments and conclusions

To test our algorithms on a higher number of processors, we ran some experiments on a Intel(R) Xeon(R) E5-2695 v4, 2.1GHz compute node with 40 processors sharing a total of 192GB memory and the codes were compiled with GCC-10.1.0. Rest of the setup was as mentioned in Sect. 2.3. We tested *mcqgHyb* and *lstoaO* using 1, 20, and 40 processors. Tables 28 and 29 show the performance of *mcqgHyb* and *lstoaO*, respectively, using up to 40 processors. While the number of instances solved using 40 threads is the same as earlier, the SGM of solution times is improved, especially for more difficult instances. On the other hand, *lstoaO40* could solve two additional instances and exhibited overall improvement in solution times.

Table 28 (Top) Comparison of *mcqgHyb1* with *mcqgHyb20* and *mcqgHyb40* on test set *TS*. *mcqgHyb* could solve 275 instances. (Bottom) Break-up of results of *mcqgHyb40* over instances of varying difficulty

Method	# Solved by		Wall time		Nodes	
(M)	M	Both	*qg*	Rel.	*qg*	Rel.
mcqgHyb20	297	275	21.47	0.38	1.5e3	1.30
mcqgHyb40	300	275	21.47	0.29	1.5e3	1.45

Time	# Solved by	Wall time		Nodes	
	Both	*qg*	Rel.	*qg*	Rel.
> 0	275	21.47	0.29	1.5e3	1.45
> 10	120	100.87	0.16	1.8e4	1.37
> 100	50	377.00	0.10	6.2e4	1.27
> 500	16	1063.29	0.06	2.1e5	1.05

Table 29 (Top) Comparison of *lstoaO1* with *lstoaO20* and *lstoaO40* on test set *TS*. *lstoaO1* could solve 307 instances. (Bottom) Break-up of results of *lstoaO40* over instances of varying difficulty

Method	# Solved by		Wall time		Nodes	
(M)	M	Both	*qg*	Rel.	*qg*	Rel.
lstoaO20	324	305	14.37	0.31	8.0e2	0.96
lstoaO40	327	305	14.37	0.24	8.0e2	0.94

Time	# Solved by	Wall time		Nodes	
	Both	*qg*	Rel.	*qg*	Rel.
> 0	305	14.37	0.31	8.0e2	0.96
> 10	126	55.56	0.14	7.3e3	0.67
> 100	30	371.39	0.07	7.0e4	0.45
> 500	11	1323.52	0.02	6.7e4	0.20

Figure 18 shows the performance profiles for these two algorithms using 1, 20, and 40 threads. *mcqgHyb40* (faster on more than 75% instances) and *lstoaO40* (faster on more than 80% instances) perform better than their respective 20-thread variants.

Next, to estimate the efficiency of our parallelization mechanism, we define and present an 'idle time ratio' = 1 - (process time)/(T*(wall clock time)), for *mcqgHyb20* and *mcqgHyb40* in Table 30. The first column categorizes the instances based on the wall clock time taken by *mcqgHyb1*. We observe that using 40 threads, mean idle times are slightly reduced (up to 0.80), more so for the difficult problems. However, a significant fraction of the wall clock time is spent by the processors waiting during the synchronization step.

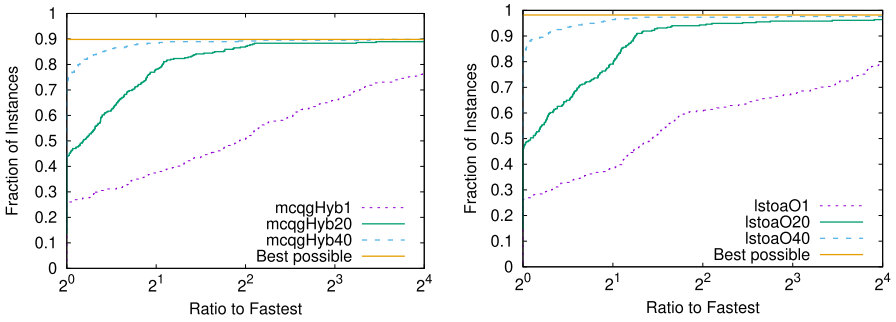

Fig. 18 Performance profiles of wall clock times taken by *mcqgHyb* (left) and *lstoaO* (right) using 1, 20 and 40 threads on test set *TS*

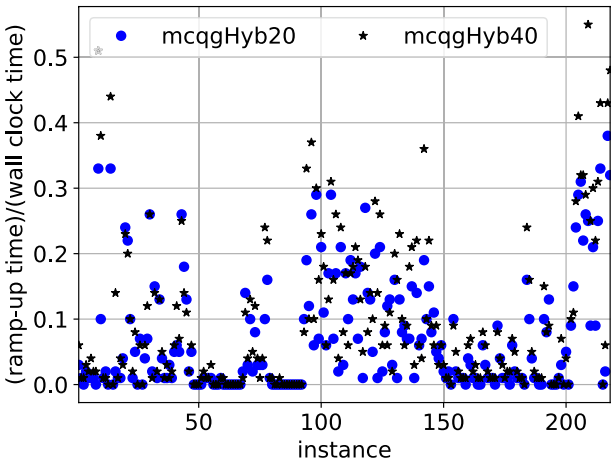

Fig. 19 Ramp-up ratios taken by *mcqgHyb* using 20 and 40 threads on 218 instances from the test set *TS*

We also analyze the ramp-up times taken by *mcqgHyb20* and *mcqgHyb40* and also compare them against the wall clock times taken to solve the instances. Figure 19 shows how the 'Ramp-up ratio' defined as (ramp-up time/wall clock time) varies across 218 instances for *mcqgHyb20* and *mcqgHyb40*. Consolidated statistics are presented in Table 31. The rows corresponding to *mcqgHyb20* and *mcqgHyb40* show the number (# Solved) of instances that could attain a ramp-up, the SGM of the ramp-up times, and the minimum and maximum ramp-up times taken. Rows corresponding to the 'Ramp-up ratio' indicates that only about 4% and 5% of the total time is spent in ramp-up for *mcqgHyb20* and *mcqgHyb40*, respectively. A shift of 0.01 was used to calculate the SGM of the ratios.

To conclude, the serial implementation of QG sees about 12% improvement in the solution time by using the proposed linearization schemes. The schemes reduce the distance between the root LP solution and the feasible region of the continuous relaxation at the root node by far greater extent than the reduction in the solution

Table 30 Idle time ratio for *mcqgHyb20* and *mcqgHyb40* on test set *TS* over instances of varying difficulty for *mcqgHyb1*

Time	# Solved by	*mcqgHyb20*			*mcqgHyb40*		
	Both	SGM	Min.	Max.	SGM	Min.	Max.
> 0	334	0.88	0.52	0.98	0.87	0.50	0.99
> 10	179	0.84	0.52	0.95	0.82	0.50	0.98
> 100	109	0.83	0.52	0.95	0.80	0.50	0.98
> 500	75	0.85	0.70	0.95	0.83	0.51	0.98

Table 31 Ramp-up times and the ramp-up ratios exhibited by *mcqgHyb* using 20 and 40 threads

	# Solved	SGM	Min.	Max.
mcqgHyb20	224	0.47	0.01	23.47
Ramp-up ratio		0.04	0.01	0.50
mcqgHyb40	221	0.75	0.01	26.25
Ramp-up ratio		0.05	0.01	0.55

time. Exploiting the univariate structure in nonlinear constraints has a more significant impact as compared to general-purpose routines. iAutomatically exploiting more nonlinear structures like separability and perspective reformulations are two promising future directions for us. Parallel extensions of the algorithms NLP-BB and QG can speed them up by about 40-50% on 40 threads. The speedup is higher for difficult instances. We see some scope of improvement here as the number of nodes processed increased by only about 50% when using 40 threads. One can improve the efficiency of the parallelization mechanism by using more opportunistic schemes. Lastly, improvements in the techniques for MILP seem to have a significant impact on the methods. MINLP solvers will gain a lot if the underneath MILP solver or the branch-and-cut implementation is improved. The scope for improvement seems especially high for the academic and open-source solvers currently available.

Appendix A

Test Instances

See Table 32

Table 32 Description of instances in the test set *TS*. The columns Set, O, and C indicate test set (TS_1 or TS_2), nonlinearity of the objective (1, otherwise 0) and the number of nonlinear constraints, respectively. Instances excluded from *TS* are: `abel`, `arki0001`, `gtm`, `harker`, `immun`, `linear`, `meanvar`, `parabol5_2_1`, `pollut`, `procsyn`, `qp2`, `qp4`, `sambal`, `sample`, `srcpm`, `turkey`, all instances with name starting from `color`, `jbearing`, `pedigree`, and 9 from `st_`

Instance	Set	O	C	Instance	Set	O	C
alan	TS_2	1	0	cvxnonsep_psig20r	TS_1	0	21
ball_mk2_10	TS_2	0	1	cvxnonsep_psig30	TS_2	1	0
ball_mk2_30	TS_2	0	1	cvxnonsep_psig30r	TS_1	0	31
ball_mk3_10	TS_2	0	1	cvxnonsep_psig40	TS_2	1	0
ball_mk3_20	TS_2	0	1	cvxnonsep_psig40r	TS_1	0	41
ball_mk3_30	TS_2	0	1	du-opt5	TS_2	1	0
ball_mk4_05	TS_2	0	1	du-opt	TS_2	1	0
ball_mk4_10	TS_2	0	1	enpro48pb	TS_2	1	1
ball_mk4_15	TS_2	0	1	enpro56pb	TS_2	1	1
batch0812	TS_2	1	1	ex1223a	TS_1	1	4
batchdes	TS_2	1	1	ex1223b	TS_2	1	4
batch	TS_2	1	1	ex1223	TS_2	1	4
batchs101006m	TS_2	1	1	ex4	TS_2	1	25
batchs121208m	TS_2	1	1	fac1	TS_2	1	0
batchs151208m	TS_2	1	1	fac2	TS_2	1	0
batchs201210m	TS_2	1	1	fac3	TS_2	1	0
clay0203h	TS_2	0	24	flay02h	TS_1	0	2
clay0203m	TS_2	0	24	flay02m	TS_1	0	2
clay0204h	TS_2	0	32	flay03h	TS_1	0	3
clay0204m	TS_2	0	32	flay03m	TS_1	0	3
clay0205h	TS_2	0	40	flay04h	TS_1	0	4
clay0205m	TS_2	0	40	flay04m	TS_1	0	4
clay0303h	TS_2	0	36	flay05h	TS_1	0	5

Table 32 (continued)

Instance	Set	O	C	Instance	Set	O	C
clay0303m	TS_2	0	36	flay05m	TS_1	0	5
clay0304h	TS_2	0	48	flay06h	TS_1	0	6
clay0304m	TS_2	0	48	flay06m	TS_1	0	6
clay0305h	TS_2	0	60	fo7_2	TS_1	0	14
clay0305m	TS_2	0	60	fo7_ar2_1	TS_1	0	14
cvxnonsep_normcon20	TS_2	0	1	fo7_ar25_1	TS_1	0	14
cvxnonsep_normcon20r	TS_1	0	20	fo7_ar3_1	TS_1	0	14
cvxnonsep_normcon30	TS_2	0	1	fo7_ar4_1	TS_1	0	14
cvxnonsep_normcon30r	TS_1	0	30	fo7_ar5_1	TS_1	0	14
cvxnonsep_normcon40	TS_2	0	1	fo7	TS_1	0	14
cvxnonsep_normcon40r	TS_1	0	40	fo8_ar2_1	TS_1	0	16
cvxnonsep_nsig20	TS_2	0	1	fo8_ar25_1	TS_1	0	16
cvxnonsep_nsig20r	TS_1	0	20	fo8_ar3_1	TS_1	0	16
cvxnonsep_nsig30	TS_2	0	1	fo8_ar4_1	TS_1	0	16
cvxnonsep_nsig30r	TS_1	0	30	fo8_ar5_1	TS_1	0	16
cvxnonsep_nsig40	TS_2	0	1	fo8	TS_1	0	16
cvxnonsep_nsig40r	TS_1	0	40	fo9_ar2_1	TS_1	0	18

Table 32 (continued)

Instance	Set	O	C	Instance	Set	O	C
cvxnonsep_pcon20	TS_2	0	1	fo9_ar25_1	TS_1	0	18
cvxnonsep_pcon20r	TS_2	0	19	fo9_ar3_1	TS_1	0	18
cvxnonsep_pcon30	TS_2	0	1	fo9_ar4_1	TS_1	0	18
cvxnonsep_pcon30r	TS_2	0	29	fo9_ar5_1	TS_1	0	18
cvxnonsep_pcon40	TS_2	0	1	fo9	TS_1	0	18
cvxnonsep_pcon40r	TS_2	0	39	gams01	TS_2	1	110
cvxnonsep_psig20	TS_2	1	0	gbd	TS_2	1	0
hybriddynamic_fixed	TS_2	1	0	rsyn0820h	TS_2	0	14
ibs2	TS_2	0	10	rsyn0820m02h	TS_2	0	28
jit1	TS_2	1	0	rsyn0820m02m	TS_1	0	28
m3	TS_1	0	6	rsyn0820m03h	TS_2	0	42
m6	TS_1	0	12	rsyn0820m03m	TS_1	0	42
m7_ar2_1	TS_1	0	14	rsyn0820m04h	TS_2	0	56
m7_ar25_1	TS_1	0	14	rsyn0820m04m	TS_1	0	56
m7_ar3_1	TS_1	0	14	rsyn0820m	TS_1	0	14
m7_ar4_1	TS_1	0	14	rsyn0830h	TS_2	0	20
m7_ar5_1	TS_1	0	14	rsyn0830m02h	TS_2	0	40
m7	TS_1	0	14	rsyn0830m02m	TS_1	0	40
meanvarx	TS_2	1	0	rsyn0830m03h	TS_2	0	60
meanvarxsc	TS_2	1	0	rsyn0830m03m	TS_1	0	60
netmod_dol1	TS_2	1	0	rsyn0830m04h	TS_2	0	80

Table 32 (continued)

Instance	Set	O	C	Instance	Set	O	C
netmod_dol2	TS_2	1	0	rsyn0830m04m	TS_1	0	80
netmod_kar1	TS_2	1	0	rsyn0830m	TS_1	0	20
netmod_kar2	TS_2	1	0	rsyn0840h	TS_2	0	28
no7_ar2_1	TS_1	0	14	rsyn0840m02h	TS_2	0	56
no7_ar25_1	TS_1	0	14	rsyn0840m02m	TS_1	0	56
no7_ar3_1	TS_1	0	14	rsyn0840m03h	TS_2	0	84
no7_ar4_1	TS_1	0	14	rsyn0840m03m	TS_1	0	84
no7_ar5_1	TS_1	0	14	rsyn0840m04h	TS_2	0	112
nvs03	TS_1	1	1	rsyn0840m04m	TS_1	0	112
nvs10	TS_2	1	2	rsyn0840m	TS_1	0	28
nvs11	TS_2	1	3	slay04h	TS_2	1	0
nvs12	TS_2	1	4	slay04m	TS_2	1	0
nvs15	TS_2	1	0	slay05h	TS_2	1	0
o7_2	TS_1	0	14	slay05m	TS_2	1	0
o7_ar2_1	TS_1	0	14	slay06h	TS_2	1	0
o7_ar25_1	TS_1	0	14	slay06m	TS_2	1	0
o7_ar3_1	TS_1	0	14	slay07h	TS_2	1	0
o7_ar4_1	TS_1	0	14	slay07m	TS_2	1	0
o7_ar5_1	TS_1	0	14	slay08h	TS_2	1	0
o7	TS_1	0	14	slay08m	TS_2	1	0
o8_ar4_1	TS_1	0	16	slay09h	TS_2	1	0
o9_ar4_1	TS_1	0	18	slay09m	TS_2	1	0
portfol_buyin	TS_2	0	2	slay10h	TS_2	1	0
portfol_card	TS_2	0	2	slay10m	TS_2	1	0

Table 32 (continued)

Instance	Set	O	C	Instance	Set	O	C
portfol_classical050_1	TS_2	0	1	smallinvDAXr1b010-011	TS_2	0	1
portfol_classical200_2	TS_2	0	1	smallinvDAXr1b020-022	TS_2	0	1
portfol_roundlot	TS_2	0	2	smallinvDAXr1b050-055	TS_2	0	1
procurement2mot	TS_1	0	12	smallinvDAXr1b100-110	TS_2	0	1
ravempb	TS_2	1	1	smallinvDAXr1b150-165	TS_2	0	1
risk2bpb	TS_2	1	0	smallinvDAXr1b200-220	TS_2	0	1
rsyn0805h	TS_2	0	3	smallinvDAXr2b010-011	TS_2	0	1
rsyn0805m02h	TS_2	0	6	smallinvDAXr2b020-022	TS_2	0	1
rsyn0805m02m	TS_1	0	6	smallinvDAXr2b050-055	TS_2	0	1
rsyn0805m03h	TS_2	0	9	smallinvDAXr2b100-110	TS_2	0	1
rsyn0805m03m	TS_1	0	9	smallinvDAXr2b150-165	TS_2	0	1
rsyn0805m04h	TS_2	0	12	smallinvDAXr2b200-220	TS_2	0	1
rsyn0805m04m	TS_1	0	12	smallinvDAXr3b010-011	TS_2	0	1
rsyn0805m	TS_1	0	3	smallinvDAXr3b020-022	TS_2	0	1
rsyn0810h	TS_2	0	6	smallinvDAXr3b050-055	TS_2	0	1
rsyn0810m02h	TS_2	0	12	smallinvDAXr3b100-110	TS_2	0	1
rsyn0810m02m	TS_1	0	12	smallinvDAXr3b150-165	TS_2	0	1
rsyn0810m03h	TS_2	0	18	smallinvDAXr3b200-220	TS_2	0	1
rsyn0810m03m	TS_1	0	18	smallinvDAXr4b010-011	TS_2	0	1
rsyn0810m04h	TS_2	0	24	smallinvDAXr4b020-022	TS_2	0	1
rsyn0810m04m	TS_1	0	24	smallinvDAXr4b050-055	TS_2	0	1
rsyn0810m	TS_1	0	6	smallinvDAXr4b100-110	TS_2	0	1
rsyn0815h	TS_2	0	11	smallinvDAXr4b150-165	TS_2	0	1

Table 32 (continued)

Instance	Set	O	C	Instance	Set	O	C
rsyn0815m02h	TS_2	0	22	smallinvDAXr4b200-220	TS_2	0	1
rsyn0815m02m	TS_1	0	22	smallinvDAXr5b010-011	TS_2	0	1
rsyn0815m03h	TS_2	0	33	smallinvDAXr5b020-022	TS_2	0	1
rsyn0815m03m	TS_1	0	33	smallinvDAXr5b050-055	TS_2	0	1
rsyn0815m04h	TS_2	0	44	smallinvDAXr5b100-110	TS_2	0	1
rsyn0815m04m	TS_1	0	44	smallinvDAXr5b150-165	TS_2	0	1
rsyn0815m	TS_1	0	11	smallinvDAXr5b200-220	TS_2	0	1
squfl010-025	TS_2	1	0	syn10m04h	TS_2	0	24
squfl010-040	TS_2	1	0	syn10m04m	TS_1	0	24
squfl010-080	TS_2	1	0	syn10m	TS_1	0	6
squfl015-060	TS_2	1	0	syn15h	TS_2	0	11
squfl015-080	TS_2	1	0	syn15m02h	TS_2	0	22
squfl020-040	TS_2	1	0	syn15m02m	TS_1	0	22
squfl020-050	TS_2	1	0	syn15m03h	TS_2	0	33
squfl020-150	TS_2	1	0	syn15m03m	TS_1	0	33
squfl025-025	TS_2	1	0	syn15m04h	TS_2	0	44
squfl025-030	TS_2	1	0	syn15m04m	TS_1	0	44
squfl025-040	TS_2	1	0	syn15m	TS_1	0	11
squfl030-100	TS_2	1	0	syn20h	TS_2	0	14
squfl030-150	TS_2	1	0	syn20m02h	TS_2	0	28
squfl040-080	TS_2	1	0	syn20m02m	TS_1	0	28
sssd08-04	TS_1	0	12	syn20m03h	TS_2	0	42
sssd12-05	TS_1	0	15	syn20m03m	TS_1	0	42
sssd15-04	TS_1	0	12	syn20m04h	TS_2	0	56

Table 32 (continued)

Instance	Set	O	C	Instance	Set	O	C
sssd15-06	TS_1	0	18	syn20m04m	TS_1	0	56
sssd15-08	TS_1	0	24	syn20m	TS_1	0	14
sssd16-07	TS_1	0	21	syn30h	TS_2	0	20
sssd18-06	TS_1	0	18	syn30m02h	TS_2	0	40
sssd18-08	TS_1	0	24	syn30m02m	TS_1	0	40
sssd20-04	TS_1	0	12	syn30m03h	TS_2	0	60
sssd20-08	TS_1	0	24	syn30m03m	TS_1	0	60
sssd22-08	TS_1	0	24	syn30m04h	TS_2	0	80
sssd25-04	TS_1	0	12	syn30m04m	TS_1	0	80
sssd25-08	TS_1	0	24	syn30m	TS_1	0	20
st_e14	TS_2	1	4	syn40h	TS_2	0	28
st_miqp2	TS_2	1	0	syn40m02h	TS_2	0	56
st_miqp3	TS_2	1	0	syn40m02m	TS_1	0	56
st_miqp4	TS_2	1	0	syn40m03h	TS_2	0	84
st_miqp5	TS_2	1	0	syn40m03m	TS_1	0	84
stockcycle	TS_2	1	0	syn40m04h	TS_2	0	112
st_test3	TS_2	1	0	syn40m04m	TS_1	0	112
st_test4	TS_2	1	0	syn40m	TS_1	0	28
st_test8	TS_2	1	0	synthes1	TS_2	1	2
st_testgr1	TS_2	1	0	synthes2	TS_1	1	3
st_testgr3	TS_2	1	0	synthes3	TS_1	1	4
st_testph4	TS_2	1	0	tls12	TS_2	0	12
syn05h	TS_2	0	3	tls2	TS_2	0	2
syn05m02h	TS_2	0	6	tls4	TS_2	0	4

Table 32 (continued)

Instance	Set	O	C	Instance	Set	O	C
syn05m02m	TS_1	0	6	tls5	TS_2	0	5
syn05m03h	TS_2	0	9	tls6	TS_2	0	6
syn05m03m	TS_1	0	9	tls7	TS_2	0	7
syn05m04h	TS_2	0	12	unitcommit1	TS_2	1	0
syn05m04m	TS_1	0	12	unitcommit_50_20_2_mod_8	TS_2	1	0
syn05m	TS_1	0	3	unitcommit_200_100_1_mod_8	TS_2	1	0
syn10h	TS_2	0	6	unitcommit_200_100_2_mod_8	TS_2	1	0
syn10m02h	TS_2	0	12	watercontamination0202	TS_2	1	0
syn10m02m	TS_1	0	12	watercontamination0202r	TS_2	1	0
syn10m03h	TS_2	0	18	watercontamination0303	TS_2	1	0
syn10m03m	TS_1	0	18	watercontamination0303r	TS_2	1	0

References

1. Abhishek, K.: Topics in mixed integer nonlinear programming. Ph.D. thesis, Lehigh University (2008)
2. Abhishek, K., Leyffer, S., Linderoth, J.: FilMINT: an outer approximation based solver for convex mixed-integer nonlinear programs. INFORMS J. Comput. **22**(4), 555–567 (2010)
3. Achterberg, T.: Conflict analysis in mixed integer programming. Discret. Optim. **4**(1), 4–20 (2007)
4. Achterberg, T.: SCIP: solving constraint integer programs. Math. Program. Comput. **1**(1), 1–41 (2009)
5. Achterberg, T., Bixby, R.E., Gu, Z., Rothberg, E., Weninger, D.: Presolve reductions in mixed integer programming. Tech. Rep. 16-44, ZIB, Takustr. 7, 14195 Berlin (2016)
6. Achterberg, T., Koch, T., Martin, A.: Branching rules revisited. Oper. Res. Lett. **33**(1), 42–54 (2005)
7. Applegate, D., Bixby, R., Cook, W., Chvátal, V.: On the solution of traveling salesman problems (1998)
8. Belotti, P.: Couenne: a user's manual. Technical report, Lehigh University, Tech. rep. (2009)
9. Belotti, P., Kirches, C., Leyffer, S., Linderoth, J., Luedtke, J., Mahajan, A.: Mixed-integer nonlinear optimization. Acta Numer **22**, 1–131 (2013)
10. Berthold, T.: A computational study of primal heuristics inside an MI(NL)P solver. J. Global Optim. **70**(1), 189–206 (2018)
11. Berthold, T., Farmer, J., Heinz, S., Perregaard, M.: Parallelization of the FICO Xpress-Optimizer. Opt. Methods Softw. **33**(3), 518–529 (2018)
12. Bixby, R., Rothberg, E.: Progress in computational mixed integer programming-a look back from the other side of the tipping point. Ann. Oper. Res. **149**(1), 37 (2007)
13. Bonami, P., Biegler, L.T., Conn, A.R., Cornuéjols, G., Grossmann, I.E., Laird, C.D., Lee, J., Lodi, A., Margot, F., Sawaya, N., et al.: An algorithmic framework for convex mixed integer nonlinear programs. Discret. Optim. **5**(2), 186–204 (2008)
14. Bonami, P., Gonçalves, J.P.: Heuristics for convex mixed integer nonlinear programs. Comput. Optim. Appl. **51**(2), 729–747 (2012)
15. Boukouvala, F., Misener, R., Floudas, C.A.: Global optimization advances in mixed-integer nonlinear programming, MINLP, and constrained derivative-free optimization. CDFO. Eur. J. Oper. Res. **252**(3), 701–727 (2016)
16. Bussieck, M.R., Drud, A.S., Meeraus, A.: MINLPLib–a collection of test models for mixed-integer nonlinear programming. INFORMS J. Comput. **15**(1), 114–119 (2003)
17. Chapman, B., Jost, G., Van Der Pas, R.: Using OpenMP: portable shared memory parallel programming, vol. 10. MIT press, Cambridge (2008)
18. Crainic, T.G., Le Cun, B., Roucairol, C.: Parallel branch-and-bound algorithms. Parallel combinatorial optimization pp. 1–28 (2006)
19. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)
20. Danna, E., Rothberg, E., Le Pape, C.: Exploring relaxation induced neighborhoods to improve MIP solutions. Math. Program. **102**(1), 71–90 (2005)
21. Dolan, E.D., Moré, J.J.: Benchmarking optimization software with performance profiles. Math. Program. **91**, 201–213 (2002)
22. Duran, M.A., Grossmann, I.E.: An outer-approximation algorithm for a class of mixed-integer nonlinear programs. Math. Program. **36**(3), 307–339 (1986)
23. Fletcher, R., Leyffer, S.: Solving mixed integer nonlinear programs by outer approximation. Math. Program. **66**(1–3), 327–349 (1994)
24. Forrest, J.: CBC MILP solver. http://www.coin-or.org/Cbc
25. Geoffrion, A.M.: Generalized benders decomposition. J. Optim. Theory Appl. **10**(4), 237–260 (1972)
26. Grama, A., Karypis, G., K, V., A, G.: Introduction to parallel computing. Addison-Wesley, Boston (2003)
27. Gupta, O.K., Ravindran, A.: Branch and bound experiments in convex nonlinear integer programming. Manage. Sci. **31**(12), 1533–1546 (1985)
28. Hart, W.E., Phillips, C.A., Eckstein, J.: PEBBL: An object-oriented framework for scalable parallel branch and bound. Tech. rep., Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States) (2013)

29. Hijazi, H., Bonami, P., Ouorou, A.: An outer-inner approximation for separable mixed-integer nonlinear programs. INFORMS J. Comput. **26**(1), 31–44 (2014)
30. Hunting, M.: The AIMMS outer approximation algorithm for MINLP. Technical Report (2011)
31. Kilinç, M., Sahinidis, N.V.: State-of-the-art in mixed-integer nonlinear programming. In: Advances and trends in optimization with engineering applications, MOS-SIAM book series on optimization, pp. 273–292. SIAM, Philadelphia (2017)
32. Kilinç, M.R.: Disjunctive cutting planes and algorithms for convex mixed integer nonlinear programming. Ph.D. thesis, University of Wisconsin-Madison (2011)
33. Kronqvist, J., Lundell, A., Westerlund, T.: The extended supporting hyperplane algorithm for convex mixed-integer nonlinear programming. J. Global Optim. **64**(2), 249–272 (2016)
34. Lima, R.M., Grossmann, I.E.: Computational advances in solving mixed integer linear programming problems. Chem. Eng. Greetings Prof. Sauro Pierucci, AIDAC **151**, 160 (2011)
35. Lin, Y., Schrage, L.: The global solver in the LINDO API. Opt. Methods Softw. **24**(4–5), 657–668 (2009)
36. Linderoth, J.T., Savelsbergh, M.W.: A computational study of search strategies for mixed integer programming. INFORMS J. Comput. **11**(2), 173–187 (1999)
37. Lundell, A., Kronqvist, J., Westerlund, T.: The supporting hyperplane optimization toolkit—a polyhedral outer approximation based convex minlp solver utilizing a single branching tree approach. Preprint, Optimization Online (2018)
38. Mahajan, A.: Presolving mixed–integer linear programs. Wiley Encyclopedia of Operations Research and Management Science (2010)
39. Mahajan, A., Leyffer, S., Linderoth, J., Luedtke, J., Munson, T.: MINOTAUR: A mixed-integer nonlinear optimization toolkit. Optimization Online **6275**, (2017)
40. Melo, W., Fampa, M., Raupp, F.: An overview of MINLP algorithms and their implementation in muriqui optimizer. Annal. Oper. Res. , 1–25 (2018)
41. Misener, R., Floudas, C.A.: Antigone: algorithms for continuous/integer global optimization of nonlinear equations. J. Global Optim. **59**(2–3), 503–526 (2014)
42. Munguía, L., Oxberry, G., Rajan, D., Shinano, Y.: Parallel PIPS-SBB: multi-level parallelism for stochastic mixed-integer programs. Comp. Opt. Appl. **73**(2), 575–601 (2019)
43. Quesada, I., Grossmann, I.E.: An LP/NLP based branch and bound algorithm for convex MINLP optimization problems. Comput. Chem. Eng. **16**(10–11), 937–947 (1992)
44. Ralphs, T., Guzelsoy, M., Mahajan, A.: SYMPHONY 5.6.9 user's manual (2015)
45. Ralphs, T., Shinano, Y., Berthold, T., Koch, T.: Parallel solvers for mixed integer linear optimization. In: Handbook of parallel constraint reasoning, pp. 283–336. Springer (2018)
46. Rockafellar, R.: Convex Analysis. Princeton University Press, Princeton, NJ (1970)
47. Sahinidis, N.V.: Baron: a general purpose global optimization software package. J. Global Optim. **8**(2), 201–205 (1996)
48. Sahinidis, N.V.: Mixed-integer nonlinear programming 2018. Opt Eng (2019)
49. Shinano, Y.: The ubiquity generator framework: 7 years of progress in parallelizing branch-and-bound. In: Operations Research Proceedings 2017, pp. 143–149. Springer (2018)
50. Shinano, Y., Achterberg, T., Berthold, T., Heinz, S., Koch, T.: ParaSCIP: a parallel extension of SCIP. In: Competence in High Performance Computing 2010, pp. 135–148. Springer (2011)
51. Shinano, Y., Berthold, T., Heinz, S.: ParaXpress: an experimental extension of the FICO Xpress-Optimizer to solve hard MIPs on supercomputers. Opt Methods Softw. **33**(3), 530–539 (2018)
52. Shinano, Y., Fujie, T.: ParaLEX: A parallel extension for the CPLEX mixed integer optimizer. In: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, pp. 97–106. Springer (2007)
53. Shinano, Y., Heinz, S., Vigerske, S., Winkler, M.: FiberSCIP–a shared memory parallelization of SCIP. INFORMS J. Comput. **30**(1), 11–30 (2017)
54. Shinano, Y., Rehfeldt, D., Galley, T.: An easy way to build parallel state-of-the-art combinatorial optimization problem solvers: A computational study on solving steiner tree problems and mixed integer semidefinite programs by using ug [SCIP-*,*]-libraries. Technical Report (2019)
55. Vigerske, S., Gleixner, A.: SCIP: Global optimization of mixed-integer nonlinear programs in a branch-and-cut framework. Opt. Methods Softw. **33**(3), 563–593 (2018)
56. Wächter, A., Biegler, L.T.: On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. Math. Program. **106**(1), 25–57 (2006)
57. Westerlund, T., Pettersson, F.: An extended cutting plane method for solving convex minlp problems. Comput. Chem. Eng. **19**, 131–136 (1995)

58. Witzig, J., Berthold, T., Heinz, S.: Experiments with conflict analysis in mixed integer programming. In: International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, pp. 211–220. Springer (2017)
59. Xu, Y., Ralphs, T.K., Ladányi, L., Saltzman, M.J.: Computational experience with a software framework for parallel integer programming. INFORMS J. Comput. **21**(3), 383–397 (2009)
60. CPLEX 12.8 user's manual (2019). https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.studio.help/pdf/usrcplex.pdf
61. FICO Xpress-Optimizer (2019). http://www.fico.com/en/Products/DMTools/xpress-overview/Pages/Xpress-Optimizer.aspx
62. Gurobi optimizer 9.0 reference manual (2019). https://www.gurobi.com/wp-content/plugins/hd_documentations/documentation/9.0/refman.pdf
63. LINDO Systems Inc (2019). https://www.lindo.com/downloads/PDF/LindoUsersManual.pdf
64. SAS/OR 15.1 user's guide mathematical programming (2019). https://support.sas.com/documentation/onlinedoc/or/151/ormpug.pdf