Extreme Strong Branching for QCQPs

Santanu S. Dey*[1], Dahye Han†[2], and Yang Wang‡[3]

[1,2]School of Industrial and Systems Engineering, Georgia Institute of Technology
[3]School of Civil and Environmental Engineering, Georgia Institute of Technology

Abstract

For mixed-integer programs (MIPs), strong branching is a highly effective variable selection method to reduce the number of nodes in the branch-and-bound algorithm. Extending it to nonlinear problems is conceptually simple but practically limited. Branching on a binary variable fixes the variable to 0 or 1, whereas branching on a continuous variable requires an additional decision to choose a branching point. Previous extensions of strong branching predefine this point and then solve $2n$ relaxations where n is the number of candidate variables to branch. We propose *extreme strong branching*, which evaluates multiple branching points per variable and jointly selects both the branching variable and point based on the objective value improvement. This approach resembles the success of strong branching for MIPs while additionally exploiting bound tightening as a byproduct. For certain types of quadratically constrained quadratic programs (QCQPs), computational experiments show that the extreme strong branching rule outperforms existing commercial solvers.

1 Introduction

The strong branching technique has been recognized as one of the most effective variable selection rules in the branch-and-bound algorithm for solving mixed-integer programs (MIPs) since its introduction [2]; see also [1, 7]. The method evaluates every fractional variable as a candidate variable to branch by fixing the candidate variable to either 0 or 1, solving relaxations at two child nodes, and evaluating the corresponding improvements in the objective function. The final branching variable is selected based on a composite score that combines the improvements from both branched child nodes, often measured by the product of the two gains in the child nodes.

Our work investigates whether the success of strong branching can be extended beyond binary variables. Although the extension to continuous variables is conceptually straightforward, its practical applicability remains limited and challenging. Unlike the case of branching on binary variables in MIP, branching on a continuous variable, known as *spatial branching* [20], requires an additional decision on the branching point.

Recent advances in spatial branch-and-bound have introduced algorithmic strategies to address the fundamental challenges of variable and branching point selections. Violation transfer rule was proposed in [21], which systematically addresses constraint violations through strategic variable selection. The author of [16] developed a geometric approach that partitions the feasible region into triangles and rectangles, enabling finer spatial decomposition and convexification. Authors of [6] designed a specialized branching strategy for quadratically constrained quadratic programs (QCQPs)

*santanu.dey@isye.gatech.edu
†dahye.han@gatech.edu
‡yang.wang@ce.gatech.edu

involving complex variables, addressing the unique difficulties in complex optimization domains. For bilinear problems, a branching rule that balances the violations across two child nodes was suggested in [10], while specifically for bilinear bipartite programs (BBPs), authors of [9] designed a branching rule that leverages second-order cone programming (SOCP) relaxations to guide variable selection decisions. More recently, a branching rule tailored for nonconvex separable piecewise linear functions was proposed in [13].

In addition, several works have adapted the strong branching technique from MIP to spatial branch-and-bound. One such variant is introduced in [4], in which a branching point is defined as a convex combination of the relaxed solution and the midpoint of a variable's bound, followed by solving $2n$ relaxations where n is the number of candidate variables to evaluate the improvement associated with the preselected branching point for each variable. In [14], the authors proposed a learning-based approach to estimate both the branching point and branching variable expected to yield the largest improvement.

Our proposed approach performs an exhaustive search for a branching point for each variable using binary search while leveraging the bound-tightening technique during this process. This approach has two main benefits:

1. Bound tightening is performed as part of the branching process. Although this resembles the spirit of the feasibility-based bound tightening or optimality-based bound tightening implemented in modern global solvers [4, 23], one key distinction is that the bound reduction here arises directly from evaluating candidate branching points and thus is a byproduct of the branching process rather than a separate preprocessing or postprocessing step.

2. Since the extreme strong branching rule considers improvement in the objective function value, it resembles closely the objective-driven rationale that makes strong branching effective in MIP. The binary search restricts the number of evaluated points, providing an efficient trade-off between the computational search effort and improvement evaluation.

2 Extreme strong branching algorithm

We consider a general nonlinear programming (NLP) of the form $\min\{c^\top x \mid x \in P\}$. For simplicity of presentation, we have considered a linear objective here, but the method holds for nonconvex objective functions as well.

Within a branch-and-bound tree, a subproblem at node k is given by:

$$\begin{aligned} &\min\ c^\top x \\ &s.t.\ x \in P_k \end{aligned} \tag{1}$$

where P_k denotes the feasible region, generally nonconvex, defined by the branching decisions taken along the path to node k. To obtain a dual bound of the problem, we consider R_k, a convex relaxation of P_k. For a branching decision on variable x_i at threshold α, we denote by:

$$R_k(x_i \le \alpha) \quad \text{and} \quad R_k(x_i \ge \alpha),$$

the convex relaxation of $P_k \cap \{x_i \le \alpha\}$ and $P_k \cap \{x_i \ge \alpha\}$, respectively.

Our proposed branching algorithm is provided in Algorithm 1 with two subroutines for binary search (Algorithm 2) and branch score (Algorithm 3). The main algorithm is based on binary search. During the search, we may obtain a bound reduction. Finally, to choose the best branching variable and branching point combination, we compute branching scores. Each element is detailed in the following sections.

2.1 Binary search

To identify the optimal branching threshold α for variable x_i, we employ a binary search over the interval defined by the current variable bounds $[\overline{lb}_i, \overline{ub}_i]$ for a fixed number of iterations. First, let us focus on the left child node problem corresponding to solving a relaxed subproblem of the form:

$$obj_L(\alpha) = \min\{c^\top x \mid x \in R_k(x_i \leq \alpha)\}.$$

Since the feasible region $R_k(x_i \leq \alpha)$ expands as α increases, $obj_L(\alpha)$ is a monotonically non-increasing function of α. The first iteration begins with $\alpha = (\overline{lb}_i + \overline{ub}_i)/2$. If the subproblem at this α is feasible and the optimal objective function value $obj_L(\alpha)$ does not exceed the tree's best incumbent objective value, obj_{ub} (i.e., the tree's upper bound), then $obj_L(\alpha)$ is recorded and α is a candidate for branching (Algorithm 2 lines 7–9). The search continues to the left for a smaller α. On the other hand, if the subproblem is infeasible or $obj_L(\alpha)$ is greater than the tree's upper bound, the search direction is reversed to the right for a larger α (Algorithm 2 lines 4–5). We note that in this case, the lower bound of x_i can be updated as further discussed in Section 2.2. This binary search is done until the binary search iteration limit.

The same binary search procedure is then applied symmetrically to the right child node corresponding to solving a relaxed subproblem of the form $obj_R(\alpha) = \min\{c^\top x \mid x \in R_k(x_i \leq \alpha)\}$. We note that $obj_R(\alpha)$ is now a monotonically non-decreasing function in α.

After completing the binary search on both the left and right sides, we have a set of candidate branching points corresponding to different α values evaluated in both left and right directions. If any α has been explored on only one side, then additional subproblems are solved so that both $obj_L(\alpha)$ and $obj_R(\alpha)$ are available for every candidate α (Algorithm 3 lines 1–2). This ensures a consistent basis for computing the strong branching scores across the potential branching points, as further discussed in Section 2.3.

Algorithm 1 Extreme Strong Branching Rule at Node k

Input: The best upper bound of the tree (obj_{ub}) and optimal objective function value to the relaxation problem at node k, $obj_p = \min\{c^T x \mid x \in R_k\}$
Output: The branching variable x_{i^*} and the branching point α^*

1: $\alpha^* := 0$; $i^* := 0$; $score^* = -\infty$;
2: **for** each $i \in \{1, \ldots, n\}$ **do**
3: $lb_L = lb_R := \overline{lb}_i$; $ub_L = ub_R = \overline{ub}_i$; $B_L = B_R := \{\}$; $O_L = O_R = \{\}$ ▷ B_L and B_R represent sets of candidate branching points obtained, and O_L and O_R represents dictionaries of branching point and optimal objective function value pairs from solving the left and right child node problems respectively
4: **for** each $iter \in \{1, \ldots, iter_{\max}\}$ **do**
5: `binary_search(`i`, `$\leq$`, `lb_L`, `ub_L`, `$\overline{ub}_i$`, `B_L`, `O_L`)`
6: **end for**
7: **for** each $iter \in \{1, \ldots, iter_{\max}\}$ **do**
8: `binary_search(`i`, `$\geq$`, `ub_R`, `lb_R`, `$\overline{lb}_i$`, `B_R`, `O_R`)`
9: **end for**
10: **for** each $\alpha \in B_L \cup B_R$ **do**
11: `branch_score(`i`, `α`)` ▷ Compute the branching score for each candidate branching points
12: **end for**
13: **end for**
14: **return** (x_{i^*}, α^*)

2.2 Bound reduction

First, we note the following conditions that we can tighten variable bounds.

Remark 1. *If the problem* $\min\{c^T x \mid x \in R_k(x_i \leq \alpha)\}$ *is infeasible or has an optimal objective value greater than the current upper bound of the branch-and-bound tree, then the lower bound of* x_i *is at least* α*.*

An analogous statement holds for the opposite branch: if $\min\{c^T x \mid x \in R_k(x_i \geq \alpha)\}$ is infeasible or has an optimal objective value greater than the current upper bound of the branch-and-bound tree, then the upper bound of x_i is at most α.

These observations provide a natural search direction for selecting the branching point α in Section 2.1. If the conditions of Remark 1 hold for the left branch, then values greater than the current α may be considered, knowing that $x_i \geq \alpha$ must hold. If the conditions of Remark 1 are not applicable, then values smaller than the current α may be considered for a potential bound tightening.

Algorithm 2 function `binary_search`$(i,\ \diamond,\ p_1,\ p_2,\ \bar{b}_i,\ B,\ O)$

$\alpha := (p_1 + p_2)/2$ ▷ Compute the midpoint of the current interval as the next branching point
obj $:= \min\{c^T x \mid x \in R_k(x_i \diamond \alpha)\}$ or ∞ if infeasible
if $obj > obj_{ub}$ **then** ▷ If the subproblem's objective value exceeds the current best upper bound
    $p_1 := \alpha$ ▷ Shift the search direction towards the other side of the feasible region per Section 2.1
    $\bar{b}_i := \alpha$ ▷ Update the corresponding bound on variable x_i per Section 2.2
else
    $p_2 := \alpha$ ▷ Otherwise, continue the search direction towards the same side of the feasible region
    $O(\alpha) := obj$ ▷ Record the optimal objective function value of this α
    $B := B \cup \{\alpha\}$ ▷ Add α to potential branching point
end if

2.3 Branching score

For each candidate threshold α generated during binary search, we compute a branching score to assess its effectiveness. Since we have explored different α's for the left and right child nodes, we might need to solve additional optimization problems to know the optimal objective function values for all α's that were considered. The score is defined as:

$$score = \max\{obj_L - obj_p, \epsilon\} \cdot \max\{obj_R - obj_p, \epsilon\},$$

where obj_p is the optimal objective function value of the relaxation at node k and $\epsilon > 0$ is a small stability constant. The branching variable and the branching point pair (i^*, α^*) with the highest branching score are selected as the branching decision at node k.

Algorithm 3 function `branch_score`$(i,\ \alpha)$

if α is a key in O_L **then** $obj_L = O_L(\alpha)$ **else** $obj_L = \min\{c^T x \mid x \in R_k(x_i \leq \alpha)\}$
if α is a key in O_R **then** $obj_R = O_R(\alpha)$ **else** $obj_R = \min\{c^T x \mid x \in R_k(x_i \geq \alpha)\}$
$score_{new} := \max\{obj_L - obj_p, \epsilon\} \cdot \max\{obj_R - obj_p, \epsilon\}$
if $score_{new} > score^*$ **then** $i^* = i$, $\alpha^* := \alpha$, and $score^* = score_{new}$

Illustrative example We illustrate the extreme strong branching rule applied for a single variable x_i with bounds $[0, 1]$. Figure 1 visualizes iteration steps taken in Algorithm 1. The algorithm iteratively evaluates candidate branching points for the left child node problem ($x_i \leq \alpha$) using binary search. Whenever points resulting in subproblems exceed the current best upper bound obj_{UB}, bounds of x_i are updated, and the remaining feasible region is considered for deciding a new candidate branching point. The same technique is applied for the right child node problem ($x_i \geq \alpha$). Finally, branching scores are calculated to select the branching variable and branching point.

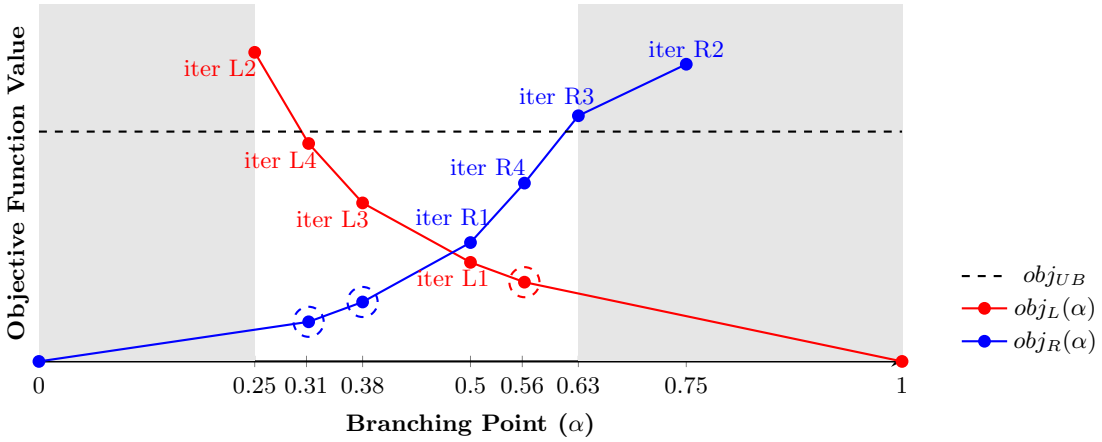

Figure 1: Illustrative example of extreme strong branching for a single variable x_i. The x-axis represents candidate branching points $\alpha \in [0, 1]$, and the y-axis shows the objective function value of the relaxed subproblem when branching at that point. The horizontal dashed line is the current best upper bound of the branch-and-bound tree, obj_{UB}. The procedure begins by exploring the left side ($x_i \leq \alpha$). At iter L1, the left side problem is solved at the midpoint $\alpha = 0.5$. Because the problem is feasible and $obj_L(0.5) < obj_{UB}$, the algorithm continues to the left for a smaller α value. Hence, at iter L2, the left side is solved for the midpoint between 0 and the previous α value 0.5 is solved. We note that $obj_L(0.25) > obj_{UB}$, which implies $x_i \geq 0.25$. Hence, the left shaded area updates the lower bound of x_i. Per the algorithm, the direction is switched so we explore a larger value of α. At iter L3 with $\alpha = (0.25 + 0.5)/2 = 0.375$, $obj_L(0.375) < obj_{UB}$ again, so at iter L4, we explore a smaller value of $\alpha = (0.25 + 0.375)/2 = 0.3125$. Once the left-side search terminates, the algorithm proceeds to the right side ($x_i \geq \alpha$). The first right side iteration also starts at $\alpha = 0.5$ and subsequently moves towards larger α values whenever the right side problem remains feasible and below the upper bound. Otherwise, we evaluate smaller α values and tighten the upper bound of the variable. Shaded regions indicate these tightened bounds. After completing both left and right side binary searches, we have a set of candidate branching points. Finally, we solve the left or right child problems for these candidate points if they have not been solved before. These are marked by dashed circles around the dot.

3 Branch-and-bound implementation

We developed a custom branch-and-bound framework, building upon the Julia package `BranchAndBound.jl` [15]. Our implementation is designed to solve a general QCQPs of the form:

$$\begin{aligned} \min_x \; & x^\top Q_0 x + p_0^\top x \\ s.t. \; & x^\top Q_k x + p_k^\top x \leq r_k, \quad \forall k \in [m] \\ & lb \leq x \leq ub \end{aligned} \tag{2}$$

where $Q_0, Q_k \in \mathbb{R}^{n \times n}$, $p_0, p_k \in \mathbb{R}^n$, and lb, ub are lower and upper bounds on the variable $x \in \mathbb{R}^n$.

To obtain a high-quality feasible solution, we first run the global solver BARON [18] (version 25.3.19) for 60 seconds and pass its best incumbent solution as the initial primal heuristic to our branch-and-bound. This practice is similar to the standard practice of providing the best-known heuristic to isolate the evaluation of the dual bound improvement [10].

The dual bound at each node is obtained by constructing a McCormick relaxation of the QCQP [17]. Specifically, each bilinear term $x_i x_j$ is replaced with an auxiliary variable w_{ij} and the following

McCormick inequalities are added:

$$\begin{aligned} w_{ij} - lb_j x_i - lb_i x_j + lb_i lb_j \geq 0 \quad & \forall i, j \in [n] \\ w_{ij} - ub_j x_i - lb_i x_j + lb_i ub_j \leq 0 \quad & \forall i, j \in [n] \\ w_{ij} - lb_j x_i - ub_i x_j + ub_i lb_j \leq 0 \quad & \forall i, j \in [n] \\ w_{ij} - ub_j x_i - ub_i x_j + ub_i ub_j \geq 0 \quad & \forall i, j \in [n]. \end{aligned} \tag{3}$$

Hence, each node in the branch-and-bound tree builds a McCormick relaxation of (2) with different local bounds of lb and ub.

We use the best-bound rule for node selection, that is, we select the node with the smallest objective function value for a minimization problem.

Once a node has been branched on, its two child nodes are added to the candidate pool while the original parent node is no longer considered. To reduce the computational overhead, the upper bound problem is solved every 10 iterations. Finally, Ipopt [22] (version 3.14.17) was used as a local NLP solver to solve (2) and HiGHS [12] (version 1.11.0) was used as an LP solver to solve the McCormick relaxation with (3). At each node, a new LP model was built without warm starting the solver with any solutions obtained from solving LPs at different nodes. The maximum iteration of the branch-and-bound algorithm was limited to 100,000.

4 Computational results

We evaluate the effectiveness of the proposed extreme strong branching rule in two settings: (i) benchmark QCQP instances from the literature [5], and (ii) an application-driven problem from structural engineering, namely the finite element model (FEM) updating problem [19, 8]. All branch-and-bound implementations were coded in the Julia language version 1.11, and experiments were executed on a personal MacBook Air equipped with an Apple M3 chip (8-core CPU) with 8 gigabytes of RAM. We report the following metrics for performance:

- Remaining optimality gap: using the best incumbent objective function value (z^*), we compute the remaining optimality gap as $\frac{|z^* - z_{lb}|}{|z^*|}$ (%) where z_{lb} denotes the lower bound on the objective function value obtained by different methods.
- Solution time in seconds: The total elapsed time is measured in wall-clock seconds. For any instance unsolved within the time limit, the maximum time (3600 seconds) allowed is reported.
- Number of nodes solved: When using the custom branch-and-bound algorithm, we record the number of nodes processed in the tree. Commercial solvers that also rely on branch-and-bound tree search provide this metric in their logs.

4.1 Benchmark QCQP instances

While the proposed rule was tested on a diverse set of QCQP instances, it proved especially effective for BBPs such as pooling problems and water management problems, which are the primary focus of our computational analysis. We considered problems with the number of variables at most 100. For these instances, to evaluate the effectiveness of the branching strategy, we compared three rules, where two rules were from previous literature on spatial branching. Both methods first consider the current infeasible relaxed solution x^*. We have not solved these with commercial solvers like Gurobi, as such solvers are highly optimized for standard instances.

- `basic`: This is a basic version of the spatial strong branching. For each candidate branching variable x_{i^*}, a branching point is considered as a weighted combination of the midpoint of a variable's bound and the current relaxed solution. A branching score is computed following Algorithm 3 and a variable with the highest branching score is selected to branch [4].
- `balance`: The branching point and the branching variable pair are selected based on the rule specifically for bilinear problems [10].
- `esb`: The branching variable and branching point are selected according to our proposed extreme strong branching rule in Algorithm 1.

Table 1 summarizes instances for which all three branching rules solved to 0.1%-optimality within the time limit. There are 35 instances of these and are grouped into four categories according to their solution time range. We report the number of instances solved (# opt) as well as the arithmetic and geometric means of the solution times (T_{ari} and T_{geo}, respectively). Overall, the `esb` rule is the only branching strategy that solves all of these instances under 100 seconds. While the `balance` rule performs competitively on some of the easier instances, we note that it requires solution time above 100 or even 1000 seconds for several of the more challenging instances. Both the arithmetic and geometric averages on the solution time of all instances also verify that `esb` rule consistently performs better. Table 2 provides the average remaining optimality gap for instances that were unsolved by at least one of the branching rules. For these more difficult instances, `esb` rule also provides the smallest remaining gap.

Detailed results are provided in Table 5 in Appendix A, where we further note that the `esb` rule explores drastically fewer nodes on average, which may not benefit much for problems requiring a relatively small-sized tree, but can outweigh the computational overhead for problems requiring a larger-sized tree by other rules.

Table 1: Average solve times for solved MINLP instances

	`basic`			`balance`			`esb`		
Time Range	# opt	T_{ari}	T_{geo}	# opt	T_{ari}	T_{geo}	# opt	T_{ari}	T_{geo}
(0, 10]	17	5.09	5.07	20	4.08	3.97	19	4.09	3.88
(10, 100]	11	48.16	43.06	11	21.20	18.54	16	22.05	19.39
(100, 1000]	6	413.75	327.06	3	163.40	143.37	0	n/a	n/a
(1000, 3600]	1	2099.35	2099.35	1	1221.00	1221.00	0	n/a	n/a
All	35	148.52	24.11	35	57.89	10.33	35	12.30	8.09

Table 2: Average remaining optimality gap for unsolved MINLP instances

# inst	`basic`	`balance`	`esb`
9	48.11%	51.21%	8.81%

4.2 Model updating problem

For FEM updating instances with 10 instances each from 16-story and 18-story structures [19, 8], we compared the entire branch-and-bound scheme with the extreme strong branching rule against directly solving it with a general-purpose commercial solver Gurobi [11] (version 12.0.1). The absolute constraint violation was set to 10^{-9} for Gurobi. Other parameters were kept as the default parameters. For both methods, we set a time limit of 3600 seconds.

Table 3 summarizes the number of instances solved to 0.1%-optimality within the time limit (# opt), along with the arithmetic and geometric means of the solution times. The performance of the branch-and-bound scheme with our proposed extreme strong branching rule solves a larger number of instances (8 out of 20), exceeding Gurobi (5 solved). Table 4 summarizes the three metrics across all instances, including instances that reached the time limit of 3600 seconds. Both the average remaining optimality gap and the number of nodes further show that the extreme strong branching rule can reach a smaller optimality gap with a smaller tree size. The entirety of the detailed computational results is provided in Table 6 in Appendix A. We have also tested our instances against BARON and Counne [3], but they did not perform better than the extreme strong branching and achieved only marginal improvements in the lower bounds with their default parameters. Although the commercial solvers have highly engineered and sophisticated internal strategies, these results show that a simple yet effective branching rule can lead to substantial performance gains without the use of any advanced convexification techniques.

Table 3: Instances solved to optimality and average time limits.

`Gurobi`			`esb`		
# opt	T_{avg}	T_{geo}	# opt	T_{avg}	T_{geo}
5	1418.81	325.99	8	1345.98	970.51

Table 4: Summary across all instances on three evaluation metrics.

	Remaining optimality gap		Solution time (sec)		Number of nodes solved	
	`Gurobi`	`esb`	`Gurobi`	`esb`	`Gurobi`	`esb`
Arithmetic mean	11.26%	7.94%	3054.70	2698.39	8607384	58
Geometric mean	3.20%	1.65%	1974.82	2130.99	4533969	44

5 Conclusion

In this paper, we introduced a new spatial branching rule, namely *extreme strong branching*. The method combines binary search with bound tightening and extends the objective-driven efficiency of strong branching from MIP to MINLP. While applicable to general MINLPs, our preliminary computational experiments show that the rule is particularly effective for bilinear bipartite problems, which are special subcases of QCQPs.

For problems with a large number of variables, our approach can have limitations, since all continuous variables are considered as candidate branching variables and the overhead computation to solve relaxation problems may outweigh the benefits. As a future direction, we would like to consider developing a reliability-based variant of extreme strong branching. Understanding why the method works particularly well for BBP is also another stream of future research.

Acknowledgment

The authors would like to gratefully acknowledge the support of grant number 2211343 from the NSF CMMI.

References

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

[2] David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. *Finding cuts in the TSP (A preliminary report)*, volume 95. Citeseer, 1995.

[3] Pietro Belotti. Couenne: a user's manual, 2009.

[4] Pietro Belotti, Jon Lee, Leo Liberti, François Margot, and Andreas Wächter. Branching and bounds tighteningtechniques for non-convex minlp. *Optimization Methods & Software*, 24(4-5):597–634, 2009.

[5] Michael R Bussieck, Arne Stolbjerg Drud, and Alexander Meeraus. Minlplib—a collection of test models for mixed-integer nonlinear programming. *INFORMS Journal on Computing*, 15(1):114–119, 2003.

[6] Chen Chen, Alper Atamtürk, and Shmuel S Oren. A spatial branch-and-cut method for nonconvex qcqp with bounded complex variables. *Mathematical Programming*, 165(2):549–577, 2017.

[7] Santanu S Dey, Yatharth Dubey, Marco Molinaro, and Prachi Shah. A theoretical and computational analysis of full strong-branching. *Mathematical Programming*, 205(1):303–336, 2024.

[8] Santanu S Dey, Dahye Han, and Yang Wang. Aggregation of bilinear bipartite equality constraints and its application to structural model updating problem. *arXiv preprint arXiv:2410.14163*, 2024.

[9] Santanu S Dey, Asteroide Santana, and Yang Wang. New socp relaxation and branching rule for bipartite bilinear programs. *Optimization and Engineering*, 20(2):307–336, 2019.

[10] Matteo Fischetti and Michele Monaci. A branch-and-cut algorithm for mixed-integer bilinear programming. *European Journal of Operational Research*, 282(2):506–514, 2020.

[11] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024.

[12] Qi Huangfu and JA Julian Hall. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10(1):119–142, 2018.

[13] Thomas Hübner, Akshay Gupte, and Steffen Rebennack. Spatial branch-and-bound for nonconvex separable piecewise linear optimization. *INFORMS Journal on Computing*, 2025.

[14] Rohit Kannan, Harsha Nagarajan, and Deepjyoti Deka. Strong partitioning and a machine learning approximation for accelerating the global optimization of nonconvex qcqps. *arXiv preprint arXiv:2301.00306*, 2022.

[15] Kibaek Kim. BranchAndBound.jl. https://github.com/kibaekkim/BranchAndBound.jl, 2020.

[16] Jeff Linderoth. A simplicial branch-and-bound algorithm for solving quadratically constrained quadratic programs. *Mathematical programming*, 103(2):251–282, 2005.

[17] Garth P McCormick. Computability of global solutions to factorable nonconvex programs: Part i—convex underestimating problems. *Mathematical programming*, 10(1):147–175, 1976.

[18] Nikolaos V Sahinidis. Baron: A general purpose global optimization software package. *Journal of global optimization*, 8(2):201–205, 1996.

[19] Trent E Schreiber, Yu Otsuki, and Yang Wang. Finite element model updating using primal-relaxed dual global optimization algorithm. *Structural Health Monitoring 2023*, 2023.

[20] Edward MB Smith and Constantinos C Pantelides. A symbolic reformulation/spatial branch-and-bound algorithm for the global optimisation of nonconvex minlps. *Computers & chemical engineering*, 23(4-5):457–478, 1999.

[21] Mohit Tawarmalani and Nikolaos V Sahinidis. Global optimization of mixed-integer nonlinear programs: A theoretical and computational study. *Mathematical programming*, 99(3):563–591, 2004.

[22] Andreas Wächter and Lorenz T Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106:25–57, 2006.

[23] Yi Zhang and Nikolaos V Sahinidis. Solving continuous and discrete nonlinear programs with baron. *Computational Optimization and Applications*, pages 1–39, 2024.

A Computational results

Table 5: Remaining optimality gap, solution time, and number of nodes solved for MINLP instances: the lowest value either by the remaining optimality gap or the solution time among the three methods is highlighted in bold; if another method reached within 10% of the lowest value, such a value is also highlighted.

	Remaining optimality gap			Solution time (sec)			Number of nodes solved		
instance	basic	balance	esb	basic	balance	esb	basic	balance	esb
pooling_adhya1pq	0.10%	0.10%	0.10%	40.40	39.06	**12.96**	201	2899	15
pooling_adhya1stp	0.10%	0.10%	0.09%	85.30	101.82	**23.65**	179	3551	11
pooling_adhya1tp	0.10%	0.10%	0.11%	355.77	**15.29**	**16.57**	1073	979	21
pooling_adhya2pq	0.09%	0.10%	0.04%	38.91	16.76	**13.87**	183	901	17
pooling_adhya2stp	0.09%	0.10%	0.08%	106.83	100.57	**22.40**	217	3375	11
pooling_adhya2tp	4.54%	0.10%	0.02%	55.63	15.09	**14.13**	215	885	17
pooling_adhya3pq	0.10%	0.10%	0.00%	90.10	**10.68**	33.81	159	265	17
pooling_adhya3stp	0.10%	0.10%	0.00%	222.75	**52.88**	64.78	153	945	11
pooling_adhya3tp	2.45%	0.10%	0.01%	262.23	**13.60**	29.02	229	405	13
pooling_adhya4pq	0.07%	0.10%	0.01%	14.67	**8.33**	15.26	15	151	5
pooling_adhya4stp	0.07%	0.09%	0.03%	27.25	**24.15**	35.36	15	493	5
pooling_adhya4tp	0.06%	0.10%	0.08%	573.45	19.89	**15.15**	421	531	5
pooling_bental4pq	0.00%	0.00%	0.00%	5.03	4.00	**3.57**	7	9	5
pooling_bental4stp	0.00%	0.00%	0.00%	5.19	**3.97**	**3.71**	7	13	5
pooling_bental4tp	0.00%	0.00%	0.00%	5.03	**3.87**	**3.57**	7	11	5
pooling_bental5pq	0.00%	0.00%	0.00%	4.39	**2.97**	**3.00**	1	1	1
pooling_bental5tp	0.00%	0.00%	0.00%	4.71	**3.04**	**3.00**	1	1	1
pooling_foulds2pq	0.00%	0.00%	0.00%	4.44	**2.90**	**3.00**	1	1	1
pooling_foulds2stp	0.00%	0.00%	0.00%	6.40	**4.03**	7.15	1	1	1
pooling_foulds2tp	0.00%	0.00%	0.00%	4.43	**2.87**	**2.85**	1	1	1
pooling_haverly1pq	0.00%	0.00%	0.00%	5.18	3.90	**3.45**	5	9	5
pooling_haverly1stp	0.00%	0.00%	0.00%	5.20	**4.00**	**3.73**	5	11	5
pooling_haverly1tp	0.00%	0.00%	0.00%	5.17	**3.90**	**3.79**	9	9	5
pooling_haverly2pq	0.00%	0.00%	0.00%	5.13	4.46	**3.61**	5	11	5
pooling_haverly2stp	0.00%	0.00%	0.00%	5.20	**3.97**	**3.66**	5	13	5
pooling_haverly2tp	0.00%	0.00%	0.00%	5.09	4.23	**3.60**	9	15	5
pooling_haverly3pq	0.00%	0.08%	0.00%	5.27	4.23	**3.54**	7	33	5
pooling_haverly3stp	0.00%	0.00%	0.00%	5.18	**3.99**	**3.72**	7	15	5
pooling_haverly3tp	0.10%	0.00%	0.00%	5.56	3.98	**3.52**	25	9	5
pooling_rt2pq	0.07%	0.10%	0.04%	37.62	**10.30**	12.40	173	627	13
pooling_rt2stp	0.10%	0.09%	0.02%	57.19	**15.51**	**14.97**	117	563	7
pooling_rt2tp	0.06%	0.05%	0.00%	50.13	**4.32**	10.01	225	33	11
wastewater02m1	0.09%	0.10%	0.01%	32.57	287.80	**5.73**	547	67287	15
wastewater02m2	0.10%	0.09%	0.00%	2099.35	**4.68**	18.49	703	217	19
wastewater04m1	0.09%	17.10%	0.03%	961.47	1221.00	**9.49**	799	112681	23
wastewater04m2	0.09%	0.10%	0.06%	3600.00	84.83	**45.41**	1087	4990	23
wastewater05m1	47.53%	56.51%	**0.08%**	3600.00	3600.00	3600.00	1521	65391	491
wastewater14m1	58.43%	56.82%	**37.43%**	3600.00	3600.00	3600.00	105	31921	609
wastewater15m1	49.74%	53.11%	**38.75%**	3600.00	3600.00	3600.00	8999	77303	1387
waterund01	2.35%	37.79%	**0.11%**	3600.00	2742.10	3600.00	11547	104743	2223
waterund08	100.00%	79.10%	**2.26%**	3600.00	3600.00	3600.00	285	37501	27
waterund11	57.10%	57.10%	**0.10%**	3600.00	3600.00	**960.04**	683	57693	189
waterund17	55.15%	63.75%	**0.39%**	3601.95	3600.00	3600.00	3095	49323	389
waterund18	62.63%	56.63%	**0.15%**	3600.09	3600.00	3600.00	3237	64743	599
Arithmetic mean	10.03%	10.90%	**1.82%**	854.51	683.02	**605.36**	825	15695	**142**
Geometric mean	0.45%	0.47%	**0.18%**	67.13.35	31.21	**24.75**	67	302	**14**

Table 6: Remaining optimality gap, solution time, and number of nodes solved by each method: the lower value either by the remaining optimality gap or the solution time achieved by two methods is highlighted in bold.

	Remaining optimality gap		Solution time (sec)		Number of nodes solved	
Instance	`Gurobi`	`esb`	`Gurobi`	`esb`	`Gurobi`	`esb`
in_19_48_1	30.13%	**27.51%**	3600.00	3600.00	9289972	105
in_19_48_2	**0.10%**	28.29%	2436.81	3600.00	6882539	105
in_19_48_3	24.15%	**14.74%**	3600.00	3600.00	14495254	93
in_19_48_4	0.05%	0.06%	**1.69**	424.67	3147	13
in_19_48_5	**0.98%**	2.10%	3600.00	3600.00	16777932	107
in_19_48_6	10.64%	**0.10%**	3600.00	3009.85	12769840	119
in_19_48_7	**5.59%**	16.30%	3600.00	3600.00	15071785	103
in_19_48_8	0.99%	**0.88%**	3600.00	3600.00	15890286	71
in_19_48_9	9.15%	**3.93%**	3600.00	3600.00	10464904	89
in_19_48_10	2.30%	**0.00%**	3600.00	1002.23	13998031	17
in_21_54_1	35.28%	**16.33%**	3600.00	3600.00	8509977	51
in_21_54_2	0.09%	0.00%	**277.49**	387.46	208340	13
in_21_54_3	38.00%	**12.39%**	3600.00	3600.00	6459876	21
in_21_54_4	6.53%	**2.20%**	3600.00	3600.00	5519562	63
in_21_54_5	38.24%	**24.77%**	3600.00	3600.00	6942537	63
in_21_54_6	0.09%	0.00%	935.95	**435.51**	633461	15
in_21_54_7	15.88%	**9.06%**	3600.00	3600.00	9509896	33
in_21_54_8	0.10%	0.01%	3442.11	**675.41**	4893953	11
in_21_54_9	5.38%	**0.10%**	3600.00	1747.07	7447186	27
in_21_54_10	1.63%	**0.07%**	3600.00	3085.60	6379203	39
Arithmetic mean	11.26%	**7.94%**	3054.70	**2698.39**	8607384	**58**
Geometric mean	3.20%	**1.65%**	1974.82	**2130.99**	4533969	**44**