Computational advances in polynomial optimization: RAPOSa, a freely available global solver

Brais González-Rodríguez*[1], Joaquín Ossorio-Castillo[3], Julio González-Díaz[1,3], Ángel M. González-Rueda[2], David R. Penas[1], and Diego Rodríguez-Martínez[3]

[1] *Department of Statistics, Mathematical Analysis and Optimization, MODESTYA Research Group and IMAT, University of Santiago de Compostela.*
[2] *Department of Mathematics, MODES Research Group and CITIC, University of A Coruña.*
[3] *ITMATI (Technological Institute for Industrial Mathematics).*

July 31, 2021

Abstract

In this paper we introduce `RAPOSa`, a global optimization solver specifically designed for (continuous) polynomial programming problems with box-constrained variables. Written entirely in `C++`, `RAPOSa` is based on the Reformulation-Linearization Technique developed by Sherali and Tuncbilek (1992) and subsequently improved in Sherali et al. (2012a), Sherali et al. (2012b) and Dalkiran and Sherali (2013). We present a description of the main characteristics of `RAPOSa` along with a thorough analysis of the impact on its performance of various enhancements discussed in the literature, such as bound tightening and SDP cuts. We also present a comparative study with three of the main state-of-the-art global optimization solvers: `BARON` (Tawarmalani and Sahinidis, 2005), `Couenne` (Belotti et al., 2009) and `SCIP` (Achterberg, 2009).

Keywords Global optimization, Polynomial programming, Reformulation-Linearization Technique (RLT).

1 Introduction

In this paper we introduce `RAPOSa` (**R**eformulation **A**lgorithm for **P**olynomial **O**ptimization - **Sa**ntiago), a new global optimization solver specifically designed for polynomial programming problems with box-constrained variables. It is based on the Reformulation-Linearization Technique (Sherali and Tuncbilek, 1992), hereafter RLT, and has been implemented in `C++`. Although it is not open source, `RAPOSa` is freely distributed and available for Linux, Windows and MacOS. It can also be run from `AMPL` (Fourer et al., 1990) and from NEOS Server (Czyzyk et al., 1998). The RLT-based scheme in `RAPOSa` solves polynomial programming problems by successive linearizations embedded into a branch-and-bound scheme. At each iteration, a linear solver must be called, and `RAPOSa` has been integrated with a wide variety of linear optimization solvers, both open source and commercial, including those available via Google OR-tools (Perron and Furnon, 2019). Further, auxiliary calls to nonlinear solvers are also performed along the branch-and-bound tree to improve the performance of the algorithm, and again both open source and commercial solvers are supported as long as they can be called from `.nl` files (Gay, 1997, 2005). More information about `RAPOSa` can be found at `http://www.itmati.com/RAPOSa/index.html`.

In conjunction with the introduction of `RAPOSa`, the other major contribution of this paper is to study the impact of different enhancements on the performance of the RLT. We discuss not only the individual impact of a series of enhancements, but also the impact of combining them. To this end, Section 4 and Section 5 contain computational analyses on the use of J-sets (Dalkiran and Sherali, 2013), warm starting of the linear relaxations, changes in the branching criterion, the introduction of bound tightening techniques (Belotti et al., 2009, 2012; Puranik and Sahinidis, 2017) and the addition of SDP cuts (Sherali et al., 2012a), among others. Interestingly, `RAPOSa` incorporates a fine-grained

*braisgonzalez.rodriguez@usc.es.

distributed parallelization of the branch-and-bound core algorithm, which delivers promising speedups as the number of available cores increases.

The most competitive configurations of `RAPOSa` according to the preceding extensive analysis are then compared to three popular state-of-the-art global optimization solvers: `BARON` (Tawarmalani and Sahinidis, 2005), `Couenne` (Belotti et al., 2009) and `SCIP` (Achterberg, 2009). The computational analysis is performed on two different test sets. The first one, DS-TS, is a set of randomly generated polynomial programming problems of different degree introduced in Dalkiran and Sherali (2016) when studying their own RLT implementation: RLT-POS.[1] The second test set, MINLPLib-TS, contains the polynomial programming problems with box-constrained and continuous variables available in MINLPLib (Bussieck et al., 2003). The main results can be summarized as follows: i) In DS-TS, all configurations of `RAPOSa` clearly outperform `BARON`, `Couenne` and `SCIP`, with the latter performing significantly worse than all the other solvers and ii) In MINLPLib-TS, differences in performance are smaller across solvers, with `SCIP` exhibiting a slightly superior performance. Importantly, the enhanced versions of `RAPOSa` are clearly superior in this test set to the baseline configuration.

The outline of the paper is as follows. In Section 2 we present a brief overview of the classic RLT scheme and different enhancements that have been introduced in recent years. In Section 3 we discuss some specifics of the implementation of `RAPOSa` and of the testing environment. In Section 4 we present some preliminary computational results, in order to define a configuration of `RAPOSa` that can be used as the baseline to assess the impact of the main enhancements, discussed in Section 5. In Section 6 we present the comparative study with `BARON`, `Couenne` and `SCIP`. Finally, we conclude in Section 7.

2 RLT: State of the art and main contributions

2.1 Brief overview of the technique

The Reformulation-Linearization Technique was originally developed in Sherali and Tuncbilek (1992). It was designed to find global optima in polynomial optimization problems of the following form:

$$\begin{array}{rll} \text{minimize} & \phi_0(\boldsymbol{x}) & \\ \text{subject to} & \phi_r(\boldsymbol{x}) \geq \beta_r, & r = 1, \ldots, R_1 \\ & \phi_r(\boldsymbol{x}) = \beta_r, & r = R_1 + 1, \ldots, R \\ & \boldsymbol{x} \in \Omega \subset \mathbb{R}^n. & \end{array} \tag{1}$$

where $N = \{1, \ldots, n\}$ denotes the set of variables, each $\phi_r(\boldsymbol{x})$ is a polynomial of degree $\delta_r \in \mathbb{N}$, $\Omega = \{\boldsymbol{x} \in \mathbb{R}^n : 0 \leq l_j \leq x_j \leq u_j < \infty, \forall j \in N\} \subset \mathbb{R}^n$ is a hyperrectangle containing the feasible region, and the degree of the problem is defined as $\delta = \max_{r \in \{0,\ldots,R\}} \delta_r$.

A multiset is a pair (S, p), in which S is a set and $p : S \to \mathbb{N}$ is a map that indicates the multiplicity of each element of S. We slightly abuse notation and use (N, δ) to denote the multiset of variables (N, p) in which $p(i) = \delta$ for each $i \in N$. For each multiset (N, p), its cardinality is defined by $|(N, p)| = \sum_{i \in N} p(i)$.

The RLT algorithm involves two main ingredients. First, the bound-factor constraints, given, for each pair of multisets J_1 and J_2 such that $J_1 \cup J_2 \subset (N, \delta)$ and $|J_1 \cup J_2| = \delta$, by

$$F_\delta(J_1, J_2) = \prod_{j \in J_1} (x_j - l_j) \prod_{j \in J_2} (u_j - x_j) \geq 0. \tag{2}$$

Note that any point in Ω satisfies all the bound-factor constraints. Second, the RLT variables, given, for each multiset $J \subset (N, \delta)$ such that $2 \leq |J| \leq \delta$, by

$$X_J = \prod_{j \in J} x_j. \tag{3}$$

Note that each multiset J can be identified with a monomial. For instance, the multiset $J = \{1, 1, 2, 3, 4, 4\}$ refers to the monomial $x_1^2 x_2 x_3 x_4^2$. Therefore, to each monomial J one can associate different bound factors of the form $J = J_1 \cup J_2$, depending on which variables are used for the lower-bound factors and which ones for the upper-bound factors. Further, monomial $J = \{1, 1, 2, 3, 4, 4\}$ also defines the RLT variable X_{112344}.

[1] Unfortunately, we could not include RLT-POS in our comparative study, since this implementation is not publicly available.

The first step of the RLT algorithm is to build a linear relaxation of the polynomial problem (1). To do this, the polynomials of the original problem are linearized by replacing all the monomials with degree greater than 1 by their corresponding RLT variable (3). This linearization is denoted by $[\cdot]_L$. Furthermore, the linearized bound-factor constraints in (2) are added to get tighter linear relaxations:

$$\begin{array}{rll} \text{minimize} & [\phi_0(\boldsymbol{x})]_L & \\ \text{subject to} & [\phi_r(\boldsymbol{x})]_L \geq \beta_r, & r = 1, \ldots, R_1 \\ & [\phi_r(\boldsymbol{x})]_L = \beta_r, & r = R_1 + 1, \ldots, R \\ & [F_\delta(J_1, J_2)]_L \geq 0, & J_1 \cup J_2 \subset (N, \delta), |J_1 \cup J_2| = \delta \\ & \boldsymbol{x} \in \Omega \subset \mathbb{R}^n. & \end{array} \tag{4}$$

Note that if constraints in (3) are added to the linear relaxation (4), the resulting problem is equivalent to problem (1). The next step of the RLT algorithm is to solve the linear relaxation in (4) to obtain a lower bound of the original polynomial problem. Next, a branch-and-bound scheme is used to find the global optimum of the polynomial problem. Since a solution of the linear relaxation that satisfies constraints in (3) is feasible to problem (1), the branching rule is usually based on violations of these constraints, which are referred to as RLT-defining identities. The convergence of this scheme to a global optimum is proven in Sherali and Tuncbilek (1992).

Along the branch-and-bound tree, the RLT algorithm obtains, and gradually increases, lower bounds for the optimal solution of the minimization problem by solving the linear relaxations. It obtains upper bounds when the solution of a linear relaxation is feasible in the original problem. The absolute and relative differences between the lower and upper bounds lead to the so called absolute and relative optimality gaps. The goal of the algorithm is to close these gaps to find the global optimum of the polynomial problem. Thus, stopping criteria often revolve around thresholds on the optimality gaps.

2.2 Enhancements of the original RLT algorithm

In this section we briefly discuss six enhancements of the basic RLT algorithm. All of them are part of the current implementation of `RAPOSa` and its impact on the performance of the algorithm is thoroughly analyzed in Section 4 and Section 5.

2.2.1 *J*-sets

This enhancement was introduced in Dalkiran and Sherali (2013), where the authors prove that it is not necessary to consider all the bound-factor constraints in the linear relaxation (4). Specifically, they prove two main results. The first one is that convergence to a global optimum is ensured even if only the bound-factor constraints associated with the monomials that appear in the original problem are included in the linear relaxation. The second result is that convergence to a global optimum is also ensured without adding the bound-factor constraints associated with monomials $J \subset J'$, provided the bound-factor constraints associated with J' are already incorporated. Equipped with these two results, the authors identify a collection of monomials, which they call J-sets, such that convergence to a global optimum is still guaranteed if only the bound-factor constraints associated with these monomials are considered.

The use of J-sets notably reduces the number of bound-factor constraints in the linear relaxation (4) (although it still grows exponentially fast as the size of the problem increases). The main benefit of this reduction is that it leads to smaller LP relaxations and, hence, the RLT algorithm requires less time in each iteration. The drawback is that the linear relaxations become less tight because they have less constraints and more iterations may be required for convergence. Nevertheless, practice has shown that the approach with J-sets is clearly superior. We corroborate this fact in the numerical analysis Section 4.1.

2.2.2 Use of an auxiliary local NLP solver

Most branch-and-bound algorithms for global optimization rely on auxiliary local solvers and the RLT scheme can also profit from them, as already discussed in Dalkiran and Sherali (2016). They proposed to call the nonlinear local solver at certain nodes of the branch-and-bound tree. In each call, the nonlinear local solver is provided with an initial solution, which is the one associated to the lower bound of the RLT algorithm at that moment.

This strategy helps to decrease the upper bound more rapidly and, hence, allows to close the optimality gap more quickly. The only drawback is the time used in the call to the nonlinear solver. Practice has shown that, in general, it is beneficial to call it only at certain nodes instead of doing so at each and every node.

2.2.3 Products of constraint factors and bound factors

This enhancement was already mentioned in Sherali and Tuncbilek (1992) when first introducing the RLT technique for polynomial programming problems. It consists of defining tighter linear programming relaxations by strengthening *constraint factors* of the form $\phi_r(\boldsymbol{x}) - \beta_r \geq 0$ of degree less than δ. More precisely, one should take products of these constraint factors and/or products of bound factors in such a way that the resulting degree is no more than δ. Similar strengthenings can also be associated to equality constraints $\phi_r(\boldsymbol{x}) = \beta_r$ of degree less than δ, by multiplying them by variables of the original problem.

Note that, although the new constraints are tighter than the original ones in the nonlinear problem, this is not necessarily so in the linear relaxations. Thus, we also preserve the original constraints to ensure that the resulting relaxations are indeed tighter and, therefore, the lower bound may increase more rapidly. Since the addition of many of these stronger constraints may complicate the solution of the linear relaxations, one should carefully balance these two opposing effects.

2.2.4 Branching criterion

The design of a branch-and-bound algorithm requires to properly define different components of the algorithm. The most widely studied ones are the search strategy in the resulting tree, pruning rules and branching criterion; refer, for instance, to Achterberg et al. (2005) and Morrison et al. (2016). In Section 5.4 we focus on the latter of these components. More precisely, we study the impact of the criterion for the selection of the branching variable on the performance of the RLT technique.

2.2.5 Bound tightening

Bound tightening techniques are at the core of most global optimization algorithms for nonlinear problems (Belotti et al., 2009, 2012; Puranik and Sahinidis, 2017). These techniques allow to reduce the search space of the algorithm by adjusting the bounds of the variables of the problem. Two main approaches have been discussed in the literature: i) Optimality-based bound tightening, OBBT, in which bounds are tightened by solving a series of relaxations of minor variations of the original problem and ii) Feasibility-based bound tightening, FBBT, in which tighter bounds are deduced directly by exploring the problem constraints. Since OBBT is computationally demanding while FBBT is not, combined schemes are often used, under which OBBT is only performed at the root node and FBBT is performed at each and every node of the branch-and-bound tree.

These techniques lead to tighter relaxations and, therefore, they do not only reduce the search space, but they also help to increase the lower bound of the optimization algorithm more rapidly. Moreover, since the resulting linear relaxations are not harder to solve than the original ones, bound tightening techniques are often very effective at improving the performance of global optimization algorithms.

2.2.6 SDP cuts

This enhancement is introduced in Sherali et al. (2012a) and consists of adding specific constraints, called positive semidefinite cuts, SDP cuts, to the linear relaxations generated along the branch-and-bound tree. These constraints are built as follows. First, a matrix of the form $\boldsymbol{M} = [\boldsymbol{y}\boldsymbol{y}^{\boldsymbol{T}}]$ is defined, where $\boldsymbol{y}$ can be any vector defined using variables and products of variables of the original problem. Since all the variables of the polynomial problem (1) are non-negative, matrix $\boldsymbol{M}$ is positive semidefinite in any feasible solution. Therefore, given a solution of the current linear relaxation, we can evaluate $\boldsymbol{M}$ at this solution, obtaining matrix $\bar{\boldsymbol{M}}$. If this matrix is not positive semidefinite, then we can identify a valid linear cut to be added to the linear relaxation. More precisely, if there is a vector $\boldsymbol{\alpha}$ such that $\boldsymbol{\alpha}^T \bar{\boldsymbol{M}} \boldsymbol{\alpha} < 0$, then constraint $\boldsymbol{\alpha}^T \boldsymbol{M} \boldsymbol{\alpha} \geq 0$ is added to the linear relaxation.

In Sherali et al. (2012a) this process is thoroughly explained and several strategies are discussed, such as different approaches to take vector $\boldsymbol{y}$, different methods to find $\boldsymbol{\alpha}$ and the number of cuts to add in each iteration.

3 Implementation and testing environment

3.1 Implementation

Each execution of RAPOSa leans on two solvers: one for solving the linear problems generated in the branch-and-bound process and one to compute upper bounds as mentioned in Section 2.2.2. The default (free) configuration of RAPOSa runs with Glop (B. De Backer, 2015) and Ipopt (Wächter and Biegler, 2006), although the best results are obtained using commercial solver Gurobi (Gurobi Optimization, 2020) instead of Glop. Currently, RAPOSa supports the following solvers:

- Linear solvers: Gurobi (Gurobi Optimization, 2020), Glop (B. De Backer, 2015), CLP (Lougee-Heimer, 2003) and lp_solve (Berkelaar et al., 2004).[2]
- Nonlinear local solvers: Ipopt (Wächter and Biegler, 2006), Knitro (Byrd et al., 2006), MINOS (Murtagh and Saunders, 1978) and CONOPT (Drud, 1985).

RAPOSa has been implemented in C++, and it is important to clarify that it connects differently to the two types of solvers. Since solving the linear problems is the most critical part of the performance, it connects with the linear solvers through their respective C++ libraries. In the case of the nonlinear local solvers, the number of calls is significantly lower, and RAPOSa sends them an intermediate .nl file with the original problem and a starting point.

Importantly, the user does not need to explicitly generate .nl files to execute RAPOSa, since it can also be run from an AMPL interface (Fourer et al., 1990).[3] Moreover, RAPOSa can also be executed on NEOS Server (Czyzyk et al., 1998).

3.2 The testing environment

All the executions reported in this paper have been performed on the supercomputer Finisterrae II, provided by Galicia Supercomputing Centre (CESGA). Specifically, we used computational nodes powered with 2 deca-core Intel Haswell 2680v3 CPUs with 128GB of RAM connected through an Infiniband FDR network, and 1TB of hard drive.

Regarding the test sets, we use two different sets of problems. The first one is taken from Dalkiran and Sherali (2016) and consists of 180 instances of randomly generated polynomial programming problems of different degrees, number of variables and density.[4] The second test set comes from the well known benchmark MINLPLib (Bussieck et al., 2003), a library of Mixed-Integer Nonlinear Programming problems. We have selected from MINLPLib those instances that are polynomial programming problems with box-constrained and continuous variables, resulting in a total of 168 instances. Hereafter we refer to the first test set as DS-TS and to the second one as MINLPLib-TS.[5]

All solvers have been run taking as stopping criterion that the relative or absolute gap is below the threshold 0.001. The time limit was set to 10 minutes in comparisons between different configurations of RAPOSa and to 1 hour in the comparison between RAPOSa and other solvers.

4 Preliminary results and RAPOSa's baseline configuration

The goal of this section is to add to RAPOSa's RLT basic implementation a minimal set of enhancements that make it robust enough to define a baseline version to assess, in Section 5, the impact of the rest of the enhancements. As we show below, the use of J-sets and an auxiliary nonlinear solver are crucial in order to be able to get some success at tackling the problems in both DS-TS and MINLPLib-TS. Importantly, we also present in this section the results of a parallelized version of the RLT technique. However, it will not be part of the rest of the analysis in this paper since it would not allow for a fair comparison with the other solvers, which do not have these parallelization capabilities.

The results in this section and in the following one are mainly illustrated by means of summary tables. Some comments are needed to explain the information shown in these tables; similar comments

[2] The connection with Glop and CLP was implemented using Google OR-tools (Perron and Furnon, 2019).

[3] Appendix B contains a short tutorial that explains how to interact with RAPOSa either directly with .nl files or through AMPL.

[4] By density we mean the proportion of monomials in the problem to the total number of possible monomials (given the degree of the problem).

[5] Instances from DS-TS can be downloaded at http://www.itmati.com/RAPOSa/files/DS-TS.zip and instances from MINLPLib-TS can be downloaded at http://www.itmati.com/RAPOSa/files/MINLPLib-TS.zip.

apply to Table 10 in Section 6, where `RAPOSa` is compared with other global optimization solvers. In the row for the number of solved instances we show between brackets the number of instances solved by at least one configuration and the total number of instances in the corresponding test set. Regarding the computation of both average and median gaps we have that i) instances solved by all configurations under study are discarded, ii) instances for which no configuration could return a gap after the time limit are also discarded, and iii) when a configuration is not able to return a lower or upper bound after the time limit, we assign to it a relative optimality gap of 10^5; the number of such instances is represented in the row called "gap = ∞". Regarding the computation of the average time, we discard instances solved by all configurations in less than 5 seconds and those not solved by any configuration in the time limit. Further, last row represents the average of the number of nodes generated by the RLT algorithm in instances solved by all configurations.

For each table discussed in the text there are two associated performance profiles (Dolan and Moré, 2002) for each test set, although, for the sake of brevity, most of them have been relegated to Appendix A. The first performance profile is for the running times and the second one for the relative optimality gaps. They contain, respectively, the instances involved in the computations of the average times and average (and median) gaps as described above.

4.1 *J*-sets

We start by evaluating the impact of the introduction of J-sets, discussed in Section 2.2.1. We can see in Table 1a that J-sets have a huge impact on the performance of the RLT technique in both tests sets: there are dramatic gains in all dimensions.

What is a bit surprising is that the configuration with J-sets does even reduce the average number of nodes explored in problems solved by both configurations. Since the version without J-sets leads to tighter relaxations, one would expect to observe a faster increase in the lower bounds and a reduction of the total number of explored nodes (at the cost of higher solve time at each node). A careful look at the individual instances reveals that the number of explored nodes turns to be quite close for most instances. Yet, there are a few "outliers" that required many more nodes to be solved for the version without J-sets, producing a large impact on the overall average. It might be worth studying further whether these outliers appeared in the configuration without J-sets by coincidence or if there is some structural reason that leads to this effect.

It is worth noting that for MINLPLib-TS the number of instances reported is 126 out of the 168 instances of this test set. This is because, for the remaining 42 instances, the version without J-sets did not even manage to solve the root node within the time limit. Not only it did not manage to return any bounds, but it ran out of time when generating the linear relaxation at the root node and we removed these instances from the analysis.

		No J-sets	With J-sets
DS-TS	solved (107/180)	36	106
	gap = ∞	144	74
	av. time	468.75	−86.47%
	av. gap	98611.11	−98.59%
	median gap	100000.00	−100.00%
	av. nodes	234.94	−45.00%
MINNLP	solved (91/126)	69	86
	gap = ∞	55	40
	av. time	407.32	−75.56%
	av. gap	86956.52	−77.00%
	median gap	100000.00	−100.00%
	av. nodes	1277.64	−38.17%

(a) J-Sets.

	No NLS	With NLS
solved (64/180)	36	64
gap = ∞	144	0
av. time	376.08	−40.72%
av. gap	100000.00	−100.00%
median gap	100000.00	−100.00%
av. nodes	229.83	−75.85%
solved (75/126)	69	70
gap = ∞	55	4
av. time	231.41	−62.99%
av. gap	94117.65	−100.00%
median gap	100000.00	−100.00%
av. nodes	2038.36	−28.05%

(b) Use of an auxiliary local NLP solver.

Table 1: Use of J-sets and auxiliary local NLP solver.

4.2 Use of an auxiliary local NLP solver

We now present the results associated to the usage of an auxiliary local NLP solver. More precisely, `Ipopt` is run at the root node and whenever the total number of solved nodes in the branch-and-bound

tree is a power of two. We tried other strategies, but we did not observe a significant impact on the resulting performance.

Table 1b shows that the inclusion of a local solver has a huge impact on the performance of the RLT technique in both test sets. Performance improves in all dimensions, and specially so at closing the gap, since the number of instances for which some bound is missing goes down from 144 to 0 in DS-TS and from 55 to 4 in MINLPLib-TS. Again, for this test set we have removed the 54 instances in which the time limit is reached when generating the linear relaxation at the root node.

We have run similar computational tests with different local NLP solvers and the results are quite robust. Therefore, the specific choice of local solver does not seem to have a significant impact on the final performance of the RLT technique.

4.3 Different LP solvers

We now check to what extent the chosen LP solver can make a difference in the performance of the RLT technique. To this end, we solve the instances in both test sets with three different solvers, two open source ones: `CLP` (Lougee-Heimer, 2003) and `Glop` (B. De Backer, 2015), and a commercial one: `Gurobi` (Gurobi Optimization, 2020).

The results in Table 2 show that `Gurobi`'s performance is superior to both `CLP` and `Glop`. On the other hand, the two open source solvers are closed to one another. These is confirmed by the performance profiles in Figure 7 in Appendix A.

		Clp	Glop	Gurobi
DS-TS	solved (128/180)	119	119	127
	gap = ∞	0	0	0
	av. time	163.15	+2.39%	−24.46%
	av. gap	0.04	+10.77%	−50.87%
	median gap	0.03	−3.54%	−61.95%
	av. nodes	885.09	−0.65%	+0.28%
MINLP	solved (102/168)	92	98	99
	gap = ∞	7	9	6
	av. time	178.94	−15.31%	−37.55%
	av. gap	2703.79	+99.96%	−49.98%
	median gap	0.57	+13.77%	+3.15%
	av. nodes	6463.11	−31.97%	−6.17%

Table 2: Use of different linear solvers.

4.4 Parallelized RLT

We now move to an enhancement of a completely different nature. `RAPOSa`, as well as all branch-and-bound algorithms, benefits from parallelization on multi-core processors. The way of exploring the solutions tree in this type of strategy makes solving each node an independent operation, suitable to be distributed through processors in the same or different computational nodes. Hence, `RAPOSa` has been parallelized, adapting the classic master-slave paradigm: a master processor guides the search in the tree, containing a queue with pending-to-solve nodes (leaves), which will be sent to a set of worker processors. In this section, we present the computational results of our parallel version of `RAPOSa`, showing the obtained speedup as a function of the number of cores.

More precisely, for each instance, `RAPOSa` was initially run in sequential mode (using one core) and a time limit of 10 minutes. Next, the parallel version was tested varying the numbers of cores and prompted to stop when each execution reached the same degree of convergence as in the non-parallel version. Thus, the analysis focuses on the improvement on the time that the parallel version needs to reach the same result as the sequential one.

Figure 1 shows the evolution of the speedup with the number of cores for DS-TS.[6] In the x-axis we represent the number of cores (master+slaves) used by `RAPOSa`, while the y-axis shows the box plot of the speedup on the 180 DS-TS instances. The red line describes the ideal speedup: the number of workers. As can be seen, the scalability of the speedup obtained by the parallel version of `RAPOSa` is

[6]We do not include the results for MINLPLib-TS because the majority of the instances are either very easy or very difficult, so the performance of the parallel version is harder to assess.

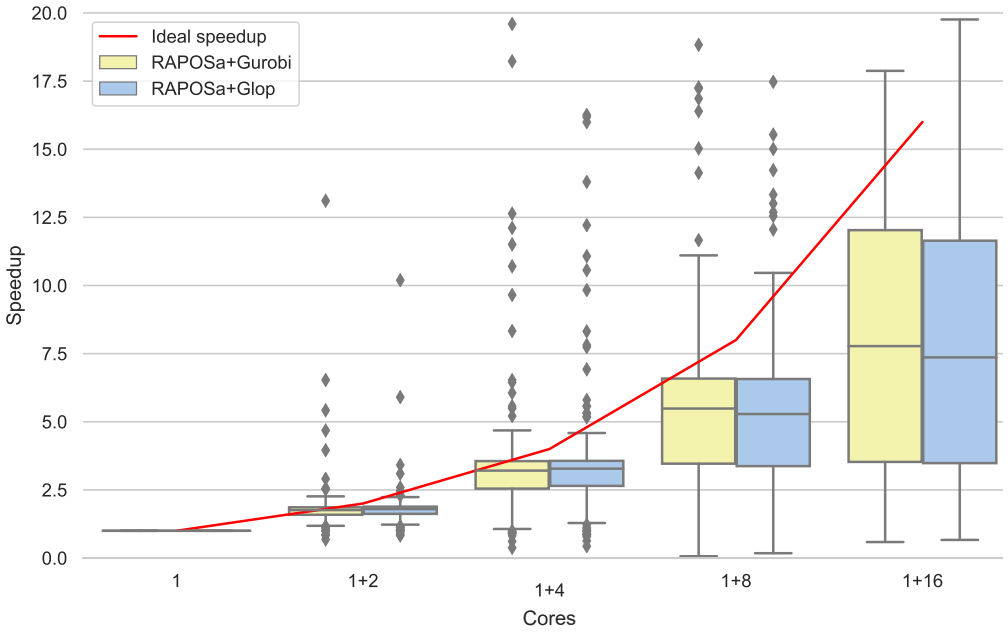


Figure 1: Speedup of the parallel version of `RAPOSa` in DS-TS.

generally good, close to the ideal one when the number of cores is small (3 and 5) or with a reasonable performance with a larger number of cores (9 and 17). Furthermore, the speedup does not seem to depend on the linear solver (`Gurobi` or `Glop`) used by RAPOSa.

5 Computational analysis of different enhancements

In this section we present a series of additional enhancements of the basic implementation of the RLT technique and try to assess not only the individual impact of each of them, but also the aggregate impact when different enhancements are combined. In order to do so, we define a baseline version of `RAPOSa` that is used as the reference for the analysis, with the different enhancements being added to this baseline. Given the results in the preceding section, this baseline configuration is set to use J-sets, `Ipopt` as the auxiliary local NLP solver,[7] and `Gurobi` as the linear one. No parallelization is used.

5.1 Warm generation of J-sets

We start with an enhancement that is essentially about efficient coding, not really about the underlying optimization algorithm.

Along the branch and bound tree, the only difference between a problem and its father problem is the upper or lower bound of a variable and the bound factors in which this variable appears. Because of this, it is reasonable to update only those bound-factor constraints that have changed instead of regenerating all the bound-factor constraints of the child node.

It is important to highlight that there are two running times involved in the process. One is the running time used to generate the bound-factor constraints, which is higher in the case that `RAPOSa` regenerates all the bound-factor constraints. The other running time is the one used to identify the bound-factor constraints that change between the parent and the child node. This time is non-existent in the case that `RAPOSa` regenerates all the bound-factor constraints. As one could expect, the number 0.00% in the last row of Table 1a reflects the fact that the warm generation of J-sets has no impact on the resulting tree.

We can see in Table 3a that the warm generation of J-sets notably improves the performance of `RAPOSa`, which is also convincingly illustrated by the performance profiles in Figure 8 in Appendix A.

[7] As we have already mentioned, we have seen in our numerical experiments that the performance of `RAPOSa` is not very sensitive to the chosen local NLP solver.

		Baseline	Warm J-sets
DS-TS	solved (136/180)	127	136
	gap = ∞	0	0
	av. time	186.94	−19.39%
	av. gap	0.02	−32.26%
	median gap	0.01	−42.47%
	av. nodes	835.72	0.00%
MINLP	solved (100/168)	99	100
	gap = ∞	6	5
	av. time	134.67	−27.77%
	av. gap	1563.76	−99.92%
	median gap	0.79	−3.12%
	av. nodes	25983.55	0.00%

(a) Warm generation of J-Sets.

	Baseline	Warm start
solved (131/180)	127	129
gap = ∞	0	0
av. time	159.53	−13.97%
av. gap	0.02	+36.01%
median gap	0.01	+73.83%
av. nodes	846.25	+7.29%
solved (100/168)	99	100
gap = ∞	6	7
av. time	134.19	−44.11%
av. gap	1.28	+123577.12%
median gap	0.74	+11.13%
av. nodes	25983.55	−3.49%

(b) Warm start on LP solver.

Table 3: Warm generation of J-sets and warm start on LP Solver.

5.2 Warm start on LP solver

As mentioned before, only a small number of bound factors are different between the child node and its father node. Because of this, it could be beneficial to feed the linear solver with information regarding the solution of the father problem (optimal solution and optimal basis).

Table 3b shows the impact of warm start with respect to the baseline configuration.[8] Differently from the preceding enhancements, the results are somewhat divided now. Warm start reduces the running time in solved instances but, on the other hand, the gap in the unsolved ones seems to deteriorate. The extremely large value of the average gap in MINLPLib-TS comes from the fact that the configuration with warm start was not able to return a gap in seven instances, whereas the baseline only failed to do so in six. This suggests that warm start may perform better in relatively easy instances, but not so well in instances that were not solved within the time limit.

5.3 Products of constraint factors and bound factors

In this section we explore the impact of strengthening the constraints of the original problem with degree less than δ by multiplying them by appropriately chosen bound factors or variables. Special care must be taken when choosing these products. The baseline includes the J-sets' enhancement and, hence, we do not want to include products that might increase the number of RLT variables in the resulting relaxations, since this might lead to a large increase in the number of bound factor constraints.

In order to avoid the above problem, our implementation of this enhancement proceeds as follows. Given a constraint of degree less than δ, we first identify what combinations of bound factors might be used to multiply the original constraint so that i) no new RLT variables are needed and ii) the resulting degree is at most δ. We then restrict attention to the combinations that involve more bound factors and distinguish between two strategies:

More in common. Each inequality constraint is multiplied by bound factor constraints that involve as many variables already present in the constraint as possible. Similarly, equality constraints are multiplied by as many variables present in the constraint as possible.

Less in common. Each inequality constraint is multiplied by bound factor constraints that involve as many variables not present in the constraint as possible. Similarly, equality constraints are multiplied by as many variables not present in the constraint as possible.

In both approaches priority is given to variables with a higher density in the original problem (present in more monomials), which showed to be a good strategy in some preliminary experiments. Importantly, we create as many new constraints as constraints we had in the original problem. Alternatively, it would be worth studying more aggressive strategies under which multiple different combinations of bound factors or variables are considered for each constraint.

[8] Currently, `RAPOSa` only supports warm start when run with `Gurobi` as linear solver.

		Baseline	More in common	Less in common
DS-TS	solved (130/180)	127	130	130
	gap = ∞	0	0	0
	av. time	153.29	−5.82%	−9.23%
	av. gap	0.02	−2.69%	−2.97%
	median gap	0.01	−0.33%	−0.33%
	av. nodes	835.72	−0.53%	−0.64%
MINLP	solved (100/168)	99	99	99
	gap = ∞	6	6	6
	av. time	128.09	+0.61%	−15.08%
	av. gap	1.26	+0.53%	+0.61%
	median gap	0.74	−0.02%	−0.02%
	av. nodes	24932.00	0.00%	−3.48%

Table 4: Products of constraint factors and bound factors.

Table 4 shows that the "More in common" approach performs slightly better in DS-TS and slightly worse in MINLPLib-TS. Interestingly, the "Less in common" approach seems to lead to slight improvements in both test sets, so it may be worth to study its impact when combined with other enhancements.

5.4 Branching criterion

In the baseline configuration, we follow the approach in Sherali and Tuncbilek (1992) and choose the branching variable involved in a maximal violation of RLT-defining identities. More precisely, given a solution $(\bar{X}, \bar{x})$ of the linear relaxation, we branch on a variable $i \in \operatorname{argmax}_{j \in N} \theta_j$, where θ_j is defined as:

$$\theta_j = \max_{t \in \{1,\dots,\delta-1\}} \max_{J \in (N,\delta):\ |J|=t} \{\left|\bar{X}_{J\cup\{j\}} - \bar{x}_j \bar{X}_J\right|\}. \tag{5}$$

In a more recent paper, Sherali et al. (2012a) apply a slightly more sophisticated criterion for variable selection, in which the maximums in the above equation are replaced by sums and also the violations associated to each variable are weighted by the minimum distance of the current value of the variable to its lower and upper bound. Here we follow a similar approach and study different criteria, where θ_j is of the form:

$$\theta_j = \sum_{t \in \{1,\dots,\delta-1\}} \sum_{J \in (N,\delta):\ |J|=t} w(j,J) \left|\bar{X}_{J\cup\{j\}} - \bar{x}_j \bar{X}_J\right|, \tag{6}$$

where the sums might be replaced by maximums as in the original approach and $w(j, J)$ represent weights that may depend on the variable and monomial at hand. We have studied a wide variety of selections for these weights and the ones that have delivered the best results are the following ones:

Constant weights. $w(j, J) = 1$ for all i and J. This corresponds with the baseline configuration when maximums are taken in Equation (6), thus recovering Equation (5). Otherwise, if sums are considered we get criterion named "Sum" in Table 5.

Coefficients. $w(j, J)$ corresponds with the sum of the absolute values of the coefficients of monomial J in the problem constraints.

Dual values. $w(j, J)$ corresponds with the sum of the absolute values of the dual values associated with the constraints of the problem containing J.

Variable range. Defining the range of a variable as the difference between its upper and lower bounds, $w(j, J)$ is taken as the quotient between the range of the variable at the current node and its range at the root node. Thus, variables whose range has been reduced less are given a higher priority.

Variable density. $w(j, J)$ is taken to be proportional to the total number of monomials in which variable j appears in the problem, so that more "active" variables are given a higher priority.

		Baseline	Sum	Coefficients	Dual values	Var. range	Var. density
DS-TS	solved (138/180)	127	135	136	136	137	134
	gap = ∞	0	0	0	0	0	0
	av. time	185.89	−17.14%	−15.75%	−9.56%	−17.49%	−7.39%
	av. gap	0.02	−12.78%	−10.55%	−7.83%	−11.84%	−11.06%
	median gap	0.01	−28.98%	−28.01%	−26.46%	−25.06%	−26.46%
	av. nodes	860.84	+19.51%	+33.95%	+33.12%	+0.82%	+69.61%
MINLP	solved (108/168)	99	105	105	104	104	104
	gap = ∞	6	5	6	6	6	5
	av. time	241.72	−51.67%	−46.34%	−48.12%	−40.72%	−41.34%
	av. gap	1516.38	−99.92%	−0.01%	+10.65%	−0.01%	−99.92%
	median gap	0.74	−19.59%	−17.78%	−9.09%	−26.56%	−19.59%
	av. nodes	8802.28	−54.72%	−45.07%	−43.99%	−55.02%	−54.37%

Table 5: Branching criterion.

Table 5 contains the results for the different criteria we have just described. Except for the baseline, which uses the maximum as in Equation (5), all other criteria use sums as in Equation (6); the reason being that criteria based on sums have shown to be remarkably superior to their "maximum" counterparts. In particular, we can see in the first two columns of the table that, just by replacing the maximum with the sum in the original branching criterion, the average computing time gets divided by two in MINLPLib-TS. In general, all criteria based on sums perform notably better than the original one. Arguably "Sum" and "Var. range" are the two most competitive ones and, by looking at the performance profiles in Figure 11 in Appendix A, it seems that "Var. range" is slightly superior, which goes along the lines of the approach taken in Sherali et al. (2012a).[9]

5.5 Bound tightening

We study the effect of different bound tightening strategies. More precisely, Table 6 represents the following approaches, along with some natural combinations of them:

OBBT root node. OBBT is run on the linear relaxation at the root node. OBBT is quite time consuming, so we limit its available time so that it does not use more than 20% of the total time available to `RAPOSa`. Since this sometimes implies that bound tightening is not applied to all variables, we prioritize tightening the upper bounds and also prioritize variables with larger ranges.

Linear FBBT. FBBT is run at all nodes on the linearized constraints of the original problem.[10]

Nonlinear FBBT. FBBT is run at all nodes on the original nonlinear constraints.

The results in Table 6 show a mild improvement on DS-TS and a very large impact on MINLPLib-TS. The fact that bound tightening has a relatively small impact on DS-TS was expected, since the generation procedure for the random instances in this test set already leads to relatively tight bounds. We can see that the nonlinear FBBT is superior to both the linear FBBT and the OBBT. Overall, the best configuration is the one combining OBBT at the root node with nonlinear FBBT at all nodes, which is a standard approach in well established global solvers. We also run some tests combining OBBT with the execution of both FBBT schemes at all nodes but, while this lead to a reduction on the average number of explored nodes, this reduction did not compensate for the additional computational overhead of running two FBBT schemes at each and every node. Additionally, we also checked if it could be beneficial to run FBBT approaches only at prespecified depths of the branch and bound tree, such as running it at nodes whose depth is a multiple of 10, but it had a detrimental effect on performance.

[9] In Sherali et al. (2012a) their weights were of the form $w(j, J) = \min\{\bar{u}_j - \bar{x}_j, \bar{x}_j - l_j\}$, which we tried as well, but they were outperformed by the ones shown in Table 5. Further, Sherali et al. (2012a) also included addends of the form $\left|\prod_{k\in J\cup\{j\}} \bar{x}_k - \bar{x}_j \bar{X}_J\right|$. In our experiments we observed no benefit from the inclusion of these terms.

[10] Our `C++` FBBT implementation builds upon the following Python implementation https://github.com/Pyomo/pyomo/tree/main/pyomo/contrib/fbbt, which is part of Pyomo's environment (Bynum et al., 2021). We also run some computational experiments using the bound tightening procedures included in `Couenne`, but the results were slightly worse.

		Baseline	OBBT root node	Lin FBBT all nodes	Nonlin FBBT all nodes	OBBT + Nonlin FBBT	OBBT + Lin FBBT
DS-TS	solved (133/180)	127	126	126	127	124	127
	gap = ∞	0	0	0	0	0	0
	av. time	165.25	−1.07%	−5.66%	−7.84%	−7.89%	−5.79%
	av. gap	0.02	+0.02%	+5.36%	+4.09%	+0.05%	+2.01%
	median gap	0.01	+1.01%	+9.38%	+4.73%	−12.29%	−2.67%
	av. nodes	919.18	−5.33%	−6.07%	−7.02%	−10.49%	−10.29%
	BT av. time	0.00	4.82	17.19	12.73	16.31	20.49
MINLP	solved (110/168)	99	97	106	109	108	102
	gap = ∞	6	5	5	5	5	5
	av. time	245.22	−11.03%	−44.78%	−68.23%	−76.91%	−46.43%
	av. gap	1493.75	−99.92%	−99.93%	−99.93%	−99.94%	−99.93%
	median gap	0.74	−20.68%	−54.06%	−55.09%	−69.06%	−54.02%
	av. nodes	26795.08	−68.22%	−70.15%	−76.00%	−77.82%	−75.69%
	BT av. time	0.00	10.28	33.06	30.38	34.76	37.43

Table 6: Bound tightening.

5.6 SDP cuts

The last enhancement we study is the introduction of SDP cuts to tighten the linear relaxations. As discussed in Section 2.2.6, the main choice of this approach is the vector $\boldsymbol{y}$ that is then used to define matrix $\boldsymbol{M} = [\boldsymbol{y}\boldsymbol{y}^{\boldsymbol{T}}]$. Importantly, the resulting cuts will involve products of the different components of $\boldsymbol{y}$ and, since we are using J-sets, we should carefully define vector $\boldsymbol{y}$ so that the resulting cuts do not lead to the inclusion of new RLT variables (which in turn would require to include additional bound factor constraints).

To minimize the impact of the above problem we proceed as follows. Note that J-sets correspond with maximal monomials with respect to set inclusion. Then, given a maximal monomial J, we define a vector $\boldsymbol{y}_J$ composed of the variables included in monomial J. Using these $\boldsymbol{y}_J$ vectors ensures that the set of maximal monomials will not increase significantly after introducing SDP cuts.[11]

We are now ready to fully describe the different approaches we have studied regarding SDP cuts, which are partially inspired in the comprehensive study developed in Sherali et al. (2012a), to which the interested reader is referred for a deeper discussion and motivation.

First, for each $\boldsymbol{y}_J$ vector we consider three possibilities to define matrix M: i) Taking $\boldsymbol{y}_J$ itself, ii) expanding it to vector $(1, \boldsymbol{y}_J)$, and iii) expanding it to a vector of the form $(1, \boldsymbol{y}_J, \ldots)$ in which, if possible, products of the variables in J are added while keeping under control the new RLT variables required by these additional products and without getting any element in M with a degree larger than δ.

For each of the above three possible definitions of the $\boldsymbol{y}$ vector, we proceed as follows: i) By default, SDP cuts are applied in all nodes and they are inherited "forever", ii) in order to save computational time in the computation of the $\boldsymbol{\alpha}$ vectors, the corresponding M matrix is divided in 10×10 overlapping submatrices (each matrix shares its first 5 rows with the preceding one), iii) for each eigenvector with a negative eigenvalue, we add the corresponding cut and iv) the procedure is repeated for each and every maximal monomial.

Table 1a shows the results of approaches $\boldsymbol{y}_J$, $(1, \boldsymbol{y}_J)$ and $(1, \boldsymbol{y}_J, \ldots)$. All of them seem to improve the performance of the algorithm, with $(1, \boldsymbol{y}_J)$ being the best of the three. Somewhat surprisingly, the average number of nodes required for the solved instances increases in DS-TS, despite using tighter relaxations at every node. This is mainly driven by a few outliers, but there may also be something structural in the randomly generated instances in DS-TS that might be worth studying further in the future. On the other hand, vectors $(1, \boldsymbol{y}_J)$ and $(1, \boldsymbol{y}_J, \ldots)$ perform very similarly in the MINLPLib-TS, the reason being that most problems in this test set are quadratic and these two vectors, by construction, coincide for quadratic (and for cubic) problems.

Given the above results, we carried out some additional experimets with vector $(1, \boldsymbol{y}_J)$. For instance, the last two columns in Table 1a represent the results when considering that cuts are only inherited to child nodes (Inh-1) and that they are also inherited to grandchildren (Inh-2). The performance

[11] If a maximal monomial corresponds with RLT variable X_{123}, when defining matrix M from vector (x_1, x_2, x_3) we get X_{11}, X_{22} and X_{33} from the relaxed diagonal elements of M. If these RLT variables were not present in the original problem, they will be added in our SDP-cuts approach. Yet, in our analysis we observed no significant impact from adding these very specific monomials.

		Baseline	$\boldsymbol{y}_J$	$(1, \boldsymbol{y}_J)$	$(1, \boldsymbol{y}_J, \ldots)$	$(1, \boldsymbol{y}_J)$ Inh-1	$(1, \boldsymbol{y}_J)$ Inh-2
DS-TS	solved (147/180)	127	123	138	134	133	136
	gap = ∞	0	0	0	0	0	0
	av. time	178.02	+30.78%	−1.77%	+26.27%	+8.35%	−1.95%
	av. gap	0.02	−8.05%	−31.84%	−19.26%	−25.23%	−33.89%
	median gap	0.01	−2.04%	−67.07%	−42.88%	−31.83%	−67.31%
	av. nodes	872.87	+120.01%	+70.54%	+66.59%	+84.01%	+73.43%
	SDP av. time	0.00	2.92	3.47	27.56	6.61	4.77
MINLP	solved (102/168)	99	99	102	102	100	102
	gap = ∞	6	7	7	6	6	6
	av. time	99.92	−15.03%	−37.65%	−37.74%	−18.08%	−37.16%
	av. gap	1.27	+123404.01%	+123401.45%	−6.98%	−6.05%	−6.48%
	median gap	0.74	−7.72%	−20.52%	−20.52%	−9.84%	−19.61%
	av. nodes	24469.35	−82.33%	−82.66%	−82.64%	−82.88%	−82.86%
	SDP av. time	0.00	5.75	8.65	9.06	15.02	13.19

Table 7: SDP cuts.

of the latter is comparable with the one with full inheritance, but no significant gain is observed. Additionally, we also studied the impact of running cycles of the form "solve → add cuts → solve → add cuts..." at each node before continuing with the branching, but in our experiments they had a detrimental effect. Similarly, we also tested configurations in which cuts being added only in nodes at prespecified depths of the branch and bound tree, such as nodes whose depth is a multiple of 10, but performance also worsened.

5.7 Combining different enhancements: RAPOSa's best configuration(s)

After having carefully discussed the individual impact of different enhancements, we now move to study how they perform when combined together, by taking the best configuration for each individual enhancement:

Warm generation of J-sets. This enhancement is included.

Warm start on LP solver. This enhancement is included.

Products of constraint factors and bound factors. We take the "less in common" approach.

Branching criterion. We consider the criterion based on variable ranges.

Bound tightening. We consider OBBT at the root node and nonlinear FBBT at all nodes.

SDP cuts. We consider the approach with vector $(1, \boldsymbol{y}_J)$, with cuts being generated in all nodes and inherited forever.

Since the number of enhancements is relatively large, we have not analyzed all the possible combinations of them. Instead, in Table 8 we represent the baseline, the combination of all enhancements together and then one column for each enhancement in which this precise enhancement has been excluded. This should help to identify conflicts or incompatibilities between enhancements.[12]

The results for DS-TS show what seems to be a conflict between warm start and SDP-cuts, since the configurations that include both of them do not even perform clearly better than the baseline. This is not completely surprising, since the behavior of warm start on this test set, as reported in Section 5.2, was already not very good. On the other hand, the version that includes all enhancements except for the warm start is clearly superior to all other configurations.

Importantly, the above conflict between warm start and SDP cuts is not present in the MNILPLib-TS library. However, the inclusion of SDP cuts seems to harm overall performance and the best configuration is precisely the one in which this enhancement is dropped.

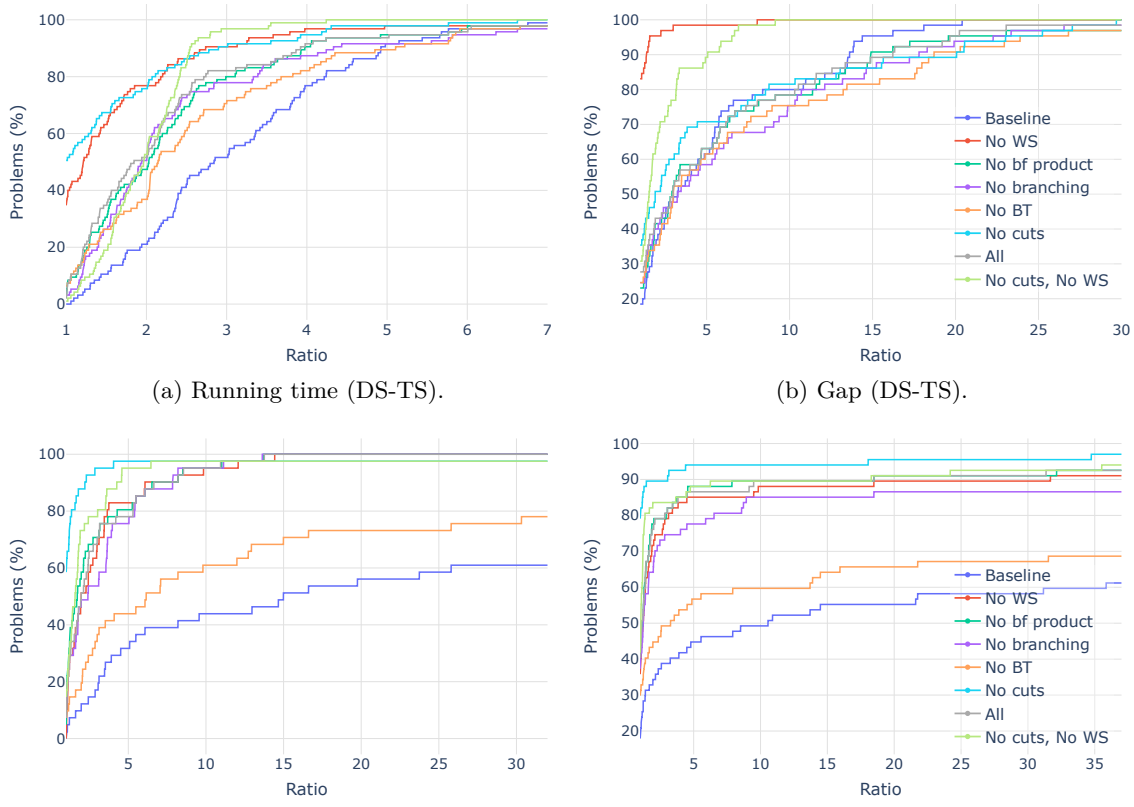
The numbers in Table 8 reveal that the enhancements under study indeed have a huge impact on the performance of the RLT technique. For DS-TS, with respect to the baseline, the configuration

[12]Since the warm generation of J-sets is just a technical enhancement about efficient coding and, as we mentioned in Section 5.1, has no impact on the resulting tree, it cannot create any conflict with other enhancements and, thus, for the sake of space, we have not included it in Table 8.

		Baseline	No WS	No Products	No Br. Crit.	No B. Tight.	No SDP cuts	All	No WS No cuts
DS-TS	solved (154/180)	127	151	130	131	131	130	133	135
	gap = ∞	0	0	0	0	0	0	0	0
	av. time	261.28	−50.14%	−16.98%	−13.74%	−10.73%	−28.58%	−19.23%	−25.41%
	av. gap	0.02	−56.74%	+3.01%	+6.75%	+9.89%	−12.36%	+1.04%	−38.69%
	median gap	0.01	−90.88%	+5.79%	+2.93%	+20.06%	−27.77%	−3.98%	−73.96%
	av. nodes	911.52	−4.72%	−5.24%	+59.05%	+4.26%	−7.17%	−5.49%	−7.04%
	BT av. time	0.00	15.56	8.53	8.89	0.00	17.00	8.35	21.85
	SDP av. time	0.00	4.93	1.99	2.13	2.17	0.00	1.98	0.00
MINLP	solved (120/168)	99	111	113	110	103	116	112	113
	gap = ∞	6	4	4	5	5	4	4	4
	av. time	328.99	−57.22%	−58.27%	−53.67%	−20.81%	−74.56%	−57.07%	−64.51%
	av. gap	2986.28	−99.97%	−99.97%	−49.99%	−49.98%	−99.97%	−99.97%	−99.97%
	median gap	0.74	−72.09%	−76.84%	−66.73%	−18.56%	−86.35%	−79.51%	−82.78%
	av. nodes	26518.92	−97.48%	−97.45%	−95.32%	−88.21%	−98.45%	−97.27%	−97.53%
	BT av. time	0.00	40.57	42.24	41.43	0.00	64.23	41.25	45.31
	SDP av. time	0.00	12.57	12.81	13.11	19.66	0.00	12.49	0.00

Table 8: Different combinations of the enhancements.

without warm start divides by two both the average time and the median gap by ten. This impact is also remarkable for MNINLPLib-TS, where the configuration without SDP cuts divides running time by three and the median gap by seven. Given these results, we decided to test a configuration in which both warm starts and SDP cuts are removed and, although his version performs fairly well, it is not the best configuration in any of the two test sets.



(a) Running time (DS-TS).

(b) Gap (DS-TS).

(c) Running time (MINLPLib-TS).

(d) Gap (MINLPLib-TS).

Figure 2: Performance profiles for different combinations of the enhancements.

In Figure 2 we have the performance profiles associated with all the configurations we have just discussed, and they confirm what could already be seen in Table 8. As expected, the baseline configuration is the worst one, but the second worse is clearly the configuration without bound tightening,

which reveals that this is the most important of the enhancements under consideration. A close look at the performance profiles suggests that the branching criterion may be the second most important one. On the other hand, the performance profiles confirm that the configuration without warm start is the best one in DS-TS and the configuration without SDP cuts is the best one in MINLPLib-TS.

Given the above results, a clear direction for future research is to get deeper into the analysis of SDP cuts. Given their promising behavior in Table 7, it would be important to understand why it has a negative impact, in MINLPLib-TS, when combined with the other enhancements.

5.8 Summary of results

To conclude this section we present in Table 9 additional joint results for the individual enhancements discussed above and the combination that performed better in each of the test sets (the version with all the enhancements except the warm start in DS-TS and the one without SDP cuts in MINLPLib-TS). Figure 3 contains the corresponding performance profiles.

		Baseline	Warm start	Constr. products	Branch crit.	Bound tight.	SDP cuts	No WS	No SDP cuts
DS-TS	solved (154/180)	127	129	130	137	124	138	151	130
	gap = ∞	0	0	0	0	0	0	0	0
	av. time	241.21	−6.25%	−3.84%	−10.68%	−3.89%	−1.05%	−50.11%	−28.58%
	av. gap	0.02	+35.67%	−2.84%	−11.62%	+0.05%	−32.01%	−56.65%	−12.34%
	median gap	0.01	+91.33%	−12.58%	−26.38%	−12.29%	−75.34%	−89.14%	−16.91%
	av. nodes	920.33	+7.36%	−0.46%	+0.79%	−10.65%	+72.75%	−4.62%	−7.08%
	BT av. time	0.00	0.00	0.00	0.00	16.31	0.00	15.56	17.00
	SDP av. time	0.00	0.00	0.00	0.00	0.00	3.47	4.93	0.00
MINLP	solved (119/168)	99	100	99	104	108	102	111	116
	gap = ∞	6	7	6	6	5	7	4	4
	av. time	293.13	−9.19%	−3.11%	−23.62%	−48.25%	−16.59%	−59.86%	−78.01%
	av. gap	2942.37	+49.98%	− 0.00%	− 0.00%	−49.99%	+49.98%	−99.97%	−99.97%
	median gap	0.74	+9.86%	−0.02%	−26.56%	−65.33%	−20.52%	−74.51%	−88.22%
	av. nodes	25451.04	−1.49%	−3.48%	−85.02%	−80.66%	−80.47%	−99.12%	−99.13%
	BT av. time	0.00	0.00	0.00	0.00	34.76	0.00	40.57	64.23
	SDP av. time	0.00	0.00	0.00	0.00	0.00	8.65	12.57	0.00

Table 9: Comparing individual enhancements with the best combinations of enhancements.

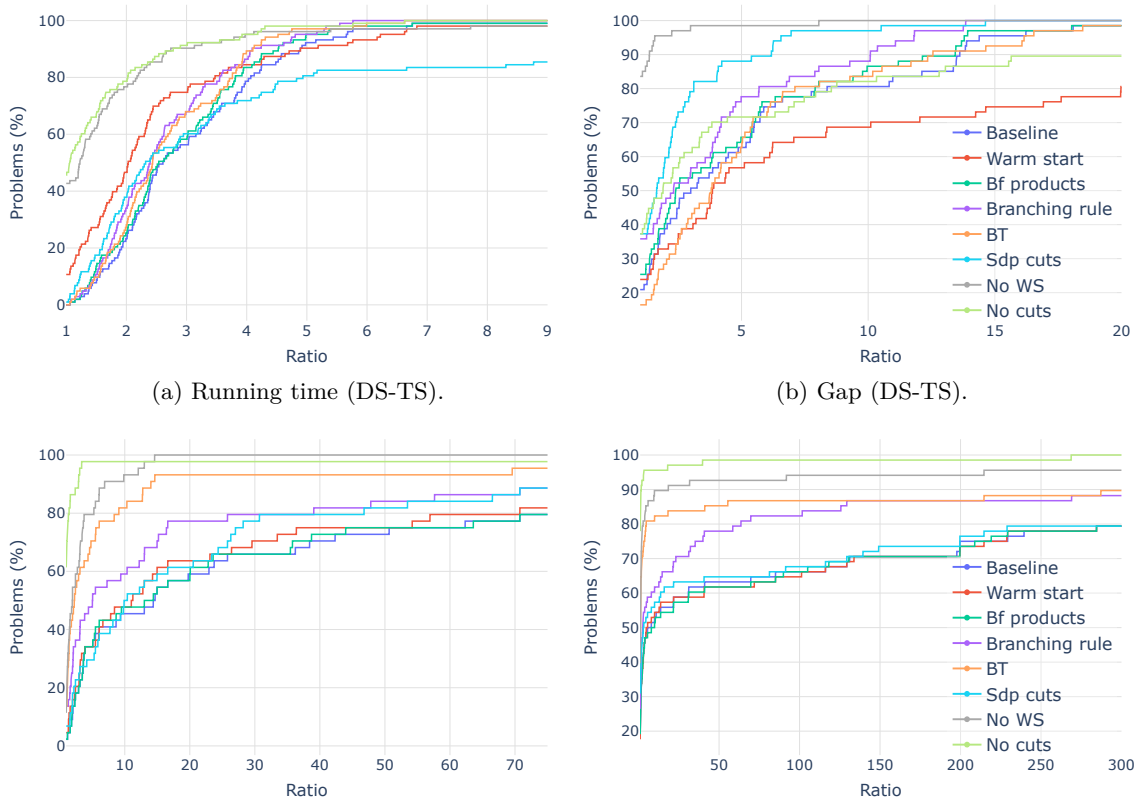
The results are fully consistent with those already discussed in the previous sections:

- The two configurations with combined enhancements perform notably better than any enhancement on its own.
- In DS-TS there is no clear winner at the individual level, since some enhancements that perform well in terms of running times perform poorly in terms of running gap and vice versa.
- In MINLPLib-TS bound tightening is clearly the enhancement that makes a larger impact on its own, with the branching criterion in second place.

6 Overall performance and comparison with other solvers

In this section we present a comparison between `RAPOSa`, `BARON`, `Couenne` and `SCIP` on the instances of DS-TS and MINLPLib-TS. `BARON`, `Couenne` and `SCIP` are three of the most popular solvers for finding global optima of nonlinear programming problems. All solvers have been run taking as stopping criterion that the relative or absolute gap is below the threshold 0.001 and with a time limit of 1 hour in each instance. `RAPOSa` was run with three different configurations: i) the baseline version considered in the previous sections, ii) a version with all the enhancements except warm start (the one with the best performance in DS-TS) and iii) a version with all the enhancements except the use of SDP cuts (the one with the best performance in MINLPLib-TS).

Table 10 contains two summaries of results, one for each test set. First, we can see that, in DS-TS, all the configurations of `RAPOSa` solved more problems than the other solvers, and the version with most solved problems is the version without warm start. We can also see that regarding running times and optimality gaps, the three configurations of `RAPOSa` are the ones that perform best. The best



(a) Running time (DS-TS).

(b) Gap (DS-TS).

(c) Running time (MINLPLib-TS).

(d) Gap (MINLPLib-TS).

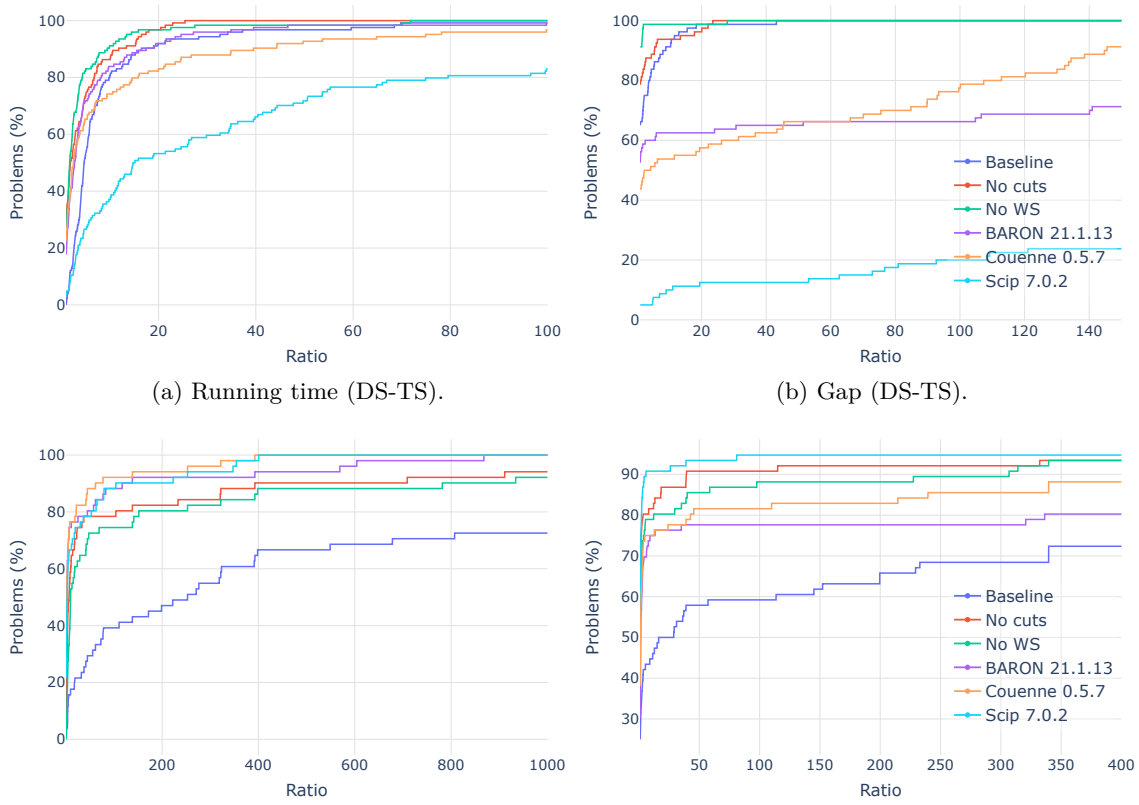
Figure 3: Performance profiles for comparing individual enhancements with the best combinations of enhancements.

		Baseline	No cuts	No WS	BARON 21.1.13	Couenne 0.5.7	SCIP 7.0.2
DS-TS	solved (172/180)	153	160	170	145	135	104
	av. time	960.586	709.421	346.989	1030.282	1220.246	2178.938
	av. gap	0.006	0.004	0.003	0.102	0.067	251.065
	median gap	0.001	0.001	0.001	0.001	0.009	0.688
MINLP	solved (132/168)	102	120	119	119	117	126
	av. time	2024.012	838.421	1114.79	651.40	907.48	553.41
	av. gap	6579.98	5195.52	6494.21	19481.05	3904.31	3896.80
	median gap	0.682	0.018	0.074	0.109	0.109	0.008

Table 10: Comparative between different configurations of `RAPOSa` and other solvers.

configuration is again the one with all the enhancements except warm start. Regarding MINLPLib-TS, SCIP is the solver that performs best. The other three solvers are quite close to each other. In particular, the configuration of RAPOSa with all the enhancements except the use of SDP cuts is not far from SCIP in terms of time, gap and number of solved instances. This same configuration of RAPOSa is significantly better than BARON in optimality gap and relatively close in average running times. When compared to Couenne, this configuration of RAPOSa is clearly better in median gap and slightly better in running times. It is worth noting that BARON, Couenne and SCIP have been tested for years on MINLPLib-TS instances and, thus, some of these solvers' enhancements may have been designed to address weaknesses identified on them. Figure 4 contains the performance profiles associated to the results reported in Table 10 and confirm the previous conclusions.

Finally, it is worth emphasizing once again the significant gain in performance exhibited by RAPOSa after the inclusion of the enhancements discussed in this paper. This is specially relevant in MINLPLib-TS since, with these changes, RAPOSa goes from being the worst solver to being highly competitive with the state-of-the-art solvers.



(a) Running time (DS-TS).

(b) Gap (DS-TS).

(c) Running time (MINLPLib-TS).

(d) Gap (MINLPLib-TS).

Figure 4: Performance profiles for comparing different configurations of RAPOSa with other solvers.

7 Conclusions and future research

In this paper we have introduced RAPOSa, a new global optimization solver specifically designed for polynomial programming problems with box-constrained variables. We have thoroughly analyzed the impact of different enhancements of the underlying RLT technique on the performance of the solver. In particular, our findings provide one more piece of evidence of the relevance of bound-tightening techniques to reduce the search space in branch and bound algorithms.

In Section 6 we compared the performance of RAPOSa with three state-of-the-art solvers an the results are already very promising, since RAPOSa's has proven to be competitive with all of them.

Yet, given that RAPOSa is still a newborn, it has a great potential for improvement and the analyses in Section 4 and Section 5 already highlight some promising directions. We conclude with a brief outline of a few of them:

Bound tightening. Given the impact on performance of bound-tightening techniques, it may be interesting to equip `RAPOSa` with more sophisticated version of both FBBT and OBBT routines, such as the ones described in Belotti (2013) and Gleixner et al. (2017).

SDP cuts. As we already mentioned in Section 5.7, a clear direction for future research is to get deeper into the analysis of SDP cuts. Given their promising behavior in Table 7, it would be important to understand why it has a negative impact, in MINLPLib-TS, when combined with the other enhancements.

Learning to branch. There is an intensive research on the use of machine learning techniques to improve the performance of the branch-and-bound algorithm, especially in integer programming (Khalil et al., 2016; Lodi and Zarpellon, 2017; Balcan et al., 2018). The integration of the insights from this research into `RAPOSa`, along with some ideas that may be specific to polynomial optimization, is definitely worth pursuing.

Integer problems. A natural avenue for `RAPOSa` is to extend its branch-and-bound scheme to allow for integer programming problems.

Acknowledgements

We are grateful to the Associate Editor and two anonymous referees for their valuable suggestions on earlier versions of the manuscript. We are grateful to Evrim Dalkiran for helpful discussions on RLT basics and past implementations and to Bissan Ghaddar for her general insights. We would also like thank to Victor Zverovich for his help handling `.nl` files. Thanks also to Raúl Alvite Pazo, Samuel Alvite Pazo, Ignacio Gómez Casares and Sergio Rodríguez Delgado for their help in implementing and validating various enhancements discussed in the paper. Finally, we are thankful to Gabriel Álvarez Castro for the maintenance of the `RAPOSa` website and to CESGA for providing their servers for the computational analyses.

This research has been funded by FEDER and the Spanish Ministry of Science and Technology through projects MTM2014-60191-JIN and MTM2017-87197-C3 (1-P and 3-P). Brais González-Rodríguez acknowledges support from the Spanish Ministry of Education through FPU grant 17/02643. David R. Penas' research is funded by the Xunta de Galicia (post-doctoral contract 2019-2022). Ángel M. González-Rueda acknowledges support from the Xunta de Galicia through the ERDF (ED431C-2020-14 and ED431G 2019/01) and "CITIC".

References

Achterberg, T. (2009): "SCIP: solving constraint integer programs," *Mathematical Programming Computation*, 1, 1–41.

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

B. De Backer, F. Didier, E. G. (2015): "Glop: An open-source linear programming solver," in *22nd International Symposium on Mathematical Programming*.

Balcan, M.-F., T. Dick, T. Sandholm, and E. Vitercik (2018): "Learning to Branch," in *Proceedings of the 35th International Conference on Machine Learning*, ed. by J. Dy and A. Krause, Stockholmsmässan, Stockholm Sweden: PMLR, vol. 80 of *Proceedings of Machine Learning Research*, 344–353.

Belotti, P. (2013): "Bound reduction using pairs of linear inequalities," *Journal of Global Optimization*, 56, 787–819.

Belotti, P., S. Cafieri, J. Lee, and L. Liberti (2012): "On the Convergence of Feasibility based Bounds Tightening," .

Belotti, P., J. Lee, L. Liberti, F. Margot, and A. Wächter (2009): "Branching and bounds tightening techniques for non-convex MINLP," *Optimization Methods and Software*, 24, 597–634.

Berkelaar, M., K. Eikland, P. Notebaert, et al. (2004): "lpsolve: Open source (mixed-integer) linear programming system," *Eindhoven U. of Technology*, 63.

Bussieck, M. R., A. S. Drud, and A. Meeraus (2003): "MINLPLib-A Collection of Test Models for Mixed-Integer Nonlinear Programming," *INFORMS J. on Computing*, 15, 114–119.

Bynum, M. L., G. A. Hackebeil, W. E. Hart, C. D. Laird, B. Nicholson, J. D. Siirola, J.-P. Watson, and D. L. Woodruff (2021): *Pyomo - Optimization Modeling in Python*, Springer International Publishing.

Byrd, R. H., J. Nocedal, and R. A. Waltz (2006): "Knitro: An Integrated Package for Nonlinear Optimization," .

Czyzyk, J., M. P. Mesnier, and J. J. More (1998): "The NEOS Server," *IEEE Journal on Computational Science and Engineering*, 5, 68–75.

Dalkiran, E. and H. D. Sherali (2013): "Theoretical filtering of RLT bound-factor constraints for solving polynomial programming problems to global optimality," *Journal of Global Optimization*, 57, 1147–1172.

——— (2016): "RLT-POS: Reformulation-Linearization Technique-based optimization software for solving polynomial programming problems," *Mathematical Programming Computation*, 8, 337–375.

Dolan, E. D. and J. J. Moré (2002): "Benchmarking optimization software with performance profiles," *Mathematical Programming*, 91, 201–213.

Drud, A. (1985): "CONOPT: A GRG code for large sparse dynamic nonlinear optimization problems," *Mathematical Programming*, 31, 153–191.

Fourer, R., D. M. Gay, and B. W. Kernighan (1990): "AMPL: A Mathematical Programing Language," *Management Science*, 36, 519–554.

Gay, D. M. (1997): "Hooking your solver to AMPL," Tech. Rep. 97-4-06, Bell Laboratories.

——— (2005): "Writing. nl files," Tech. Rep. 2005-7907P, Sandia National Laboratories.

Gleixner, A. M., T. Berthold, B. Müller, and S. Weltge (2017): "Three enhancements for optimization-based bound tightening," *J. Global Optimization*, 67, 731–757.

Gurobi Optimization (2020): "Gurobi Optimizer Reference Manual," Available at: http://www.gurobi.com.

Khalil, E. B., P. L. Bodic, L. Song, G. L. Nemhauser, and B. Dilkina (2016): "Learning to Branch in Mixed Integer Programming," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, ed. by D. Schuurmans and M. P. Wellman, AAAI Press, 724–731.

Lodi, A. and G. Zarpellon (2017): "On learning and branching: a survey," *TOP*, 207–236.

Lougee-Heimer, R. (2003): "The Common Optimization INterface for Operations Research: Promoting open-source software in the operations research community," *IBM Journal of Research and Development*, 47, 57–66.

Morrison, D. R., S. H. Jacobson, J. J. Sauppe, and E. C. Sewell (2016): "Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning," *Discrete Optimization*, 19, 79–102.

Murtagh, B. A. and M. A. Saunders (1978): "Large-scale linearly constrained optimization," *Mathematical Programming*, 14, 41–72.

Perron, L. and V. Furnon (2019): "OR-Tools," Available at: https://developers.google.com/optimization/.

Puranik, Y. and N. V. Sahinidis (2017): "Domain reduction techniques for global NLP and MINLP optimization," *Constraints*, 22, 338–376.

Sherali, H. D., E. Dalkiran, and J. Desai (2012a): "Enhancing RLT-based relaxations for polynomial programming problems via a new class of v-semidefinite cuts," *Computational Optimization and Applications*, 52, 483–506.

Sherali, H. D., E. Dalkiran, and L. Liberti (2012b): "Reduced RLT representations for nonconvex polynomial programming problems," *Journal of Global Optimization*, 52, 447–469.

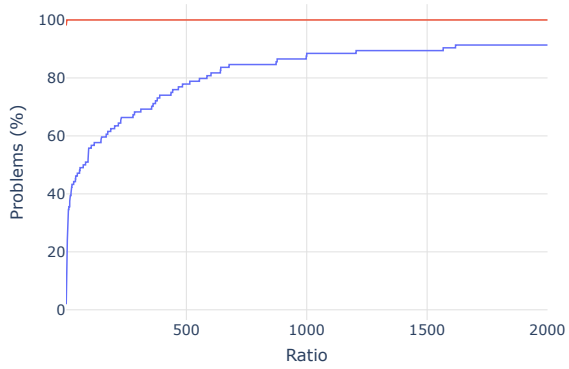
Sherali, H. D. and C. H. Tuncbilek (1992): "A global optimization algorithm for polynomial programming problems using a Reformulation-Linearization Technique," *Journal of Global Optimization*, 103, 225–249.

Tawarmalani, M. and N. Sahinidis (2005): "A polyhedral branch-and-cut approach to global optimization," *Mathematical Programming*, 2, 101–112.

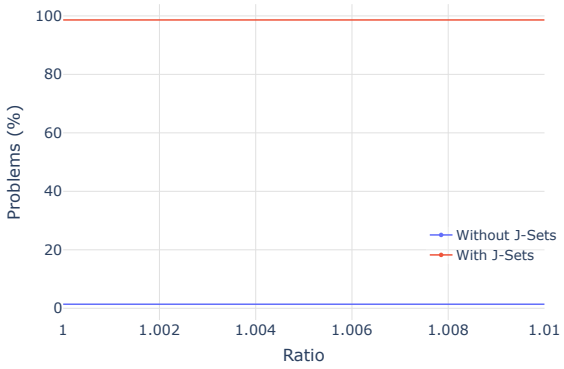
Wächter, A. and L. T. Biegler (2006): "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical Programming*, 106, 25–57.

A Performance profiles for the computational experiments in Sections 4 and 5

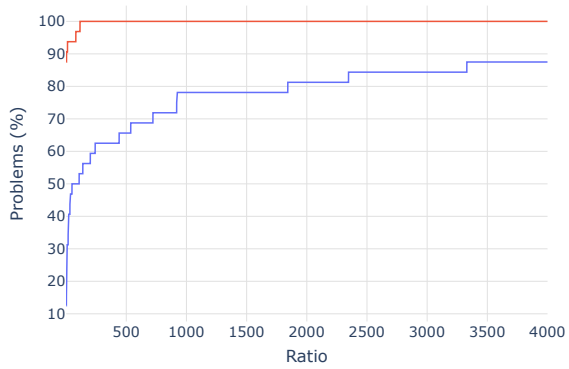
In this Appendix we present all performance profiles associated with the computational analyisis of the different enhancements of the basic RLT discussed in Section 4 and Section 5.



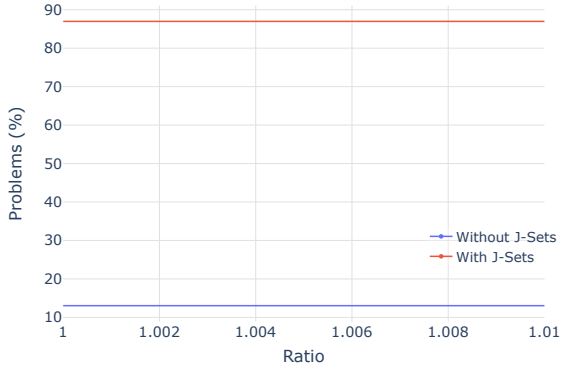
(a) Running time (DS-TS).



(b) Gap (DS-TS).

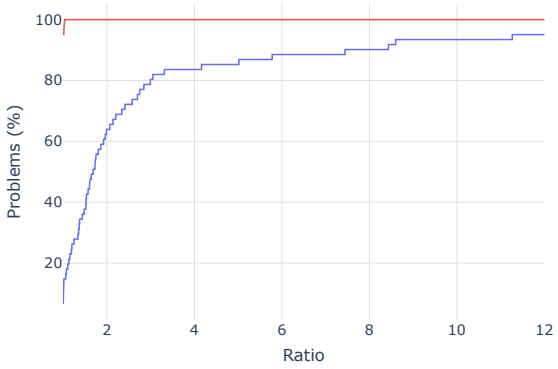


(c) Running time (MINLPLib-TS).

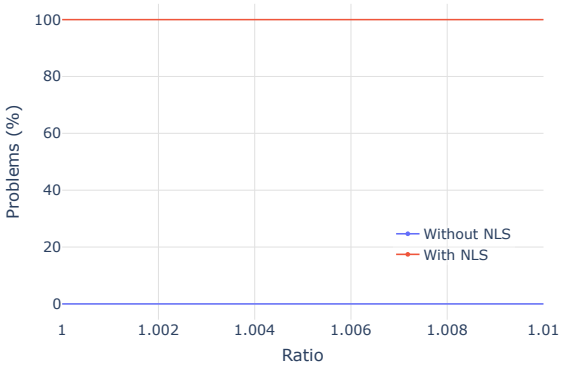


(d) Gap (MINLPLib-TS).

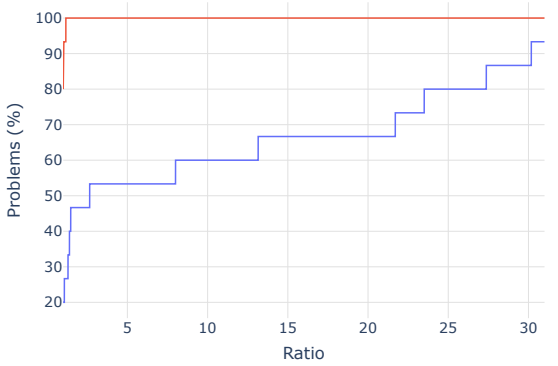
Figure 5: Performance profiles for the J-Sets.



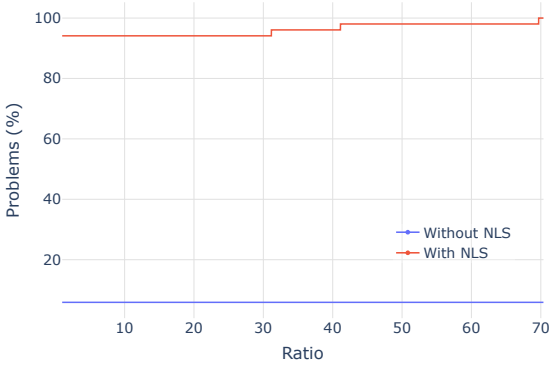
(a) Running time (DS-TS).



(b) Gap (DS-TS).

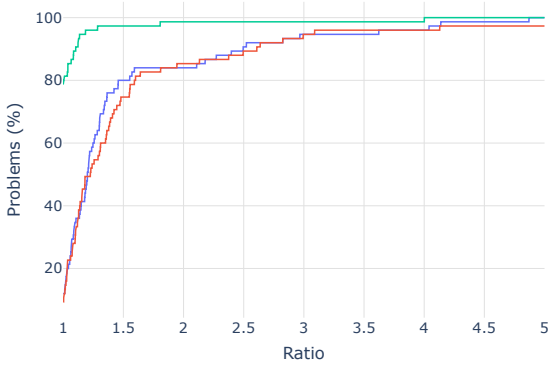


(c) Running time (MINLPLib-TS).

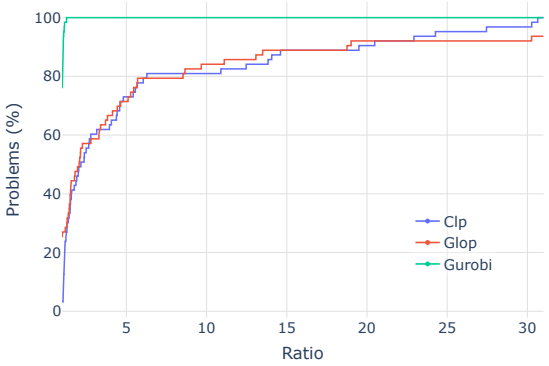


(d) Gap (MINLPLib-TS).

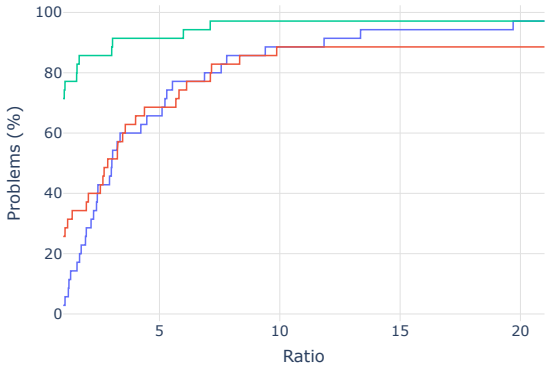
Figure 6: performance profiles for the use of an auxiliary nonlinear solver.



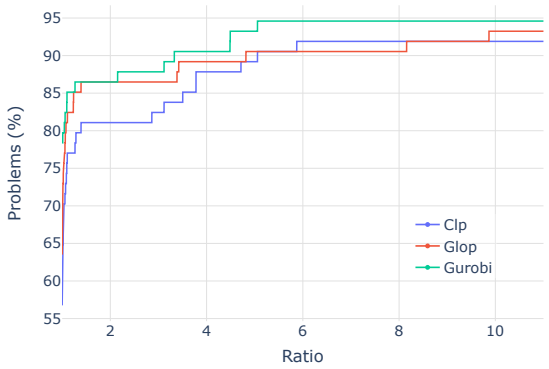
(a) Running time (DS-TS).



(b) Gap (DS-TS).

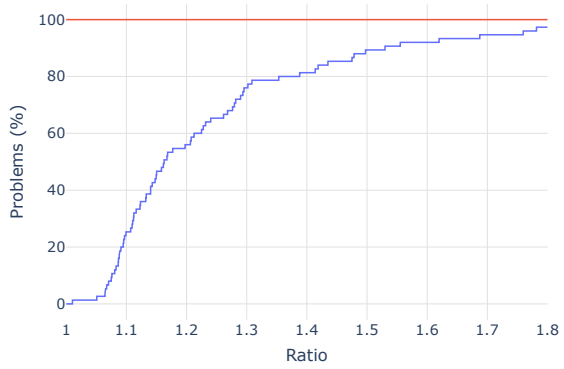


(c) Running time (MINLPLib-TS).

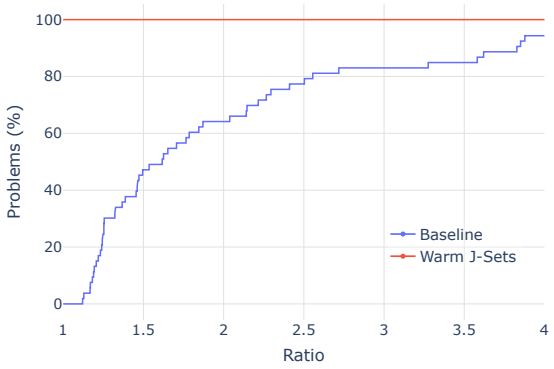


(d) Gap (MINLPLib-TS).

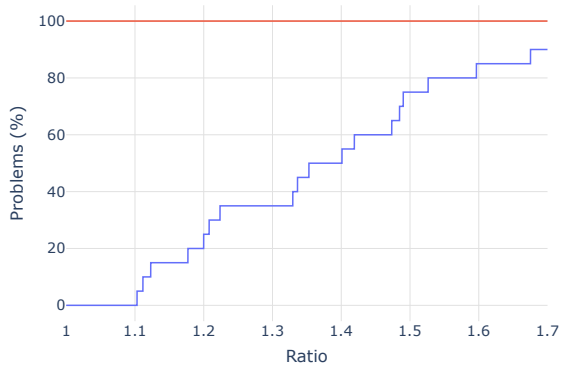
Figure 7: Performance profiles for the different linear solvers.



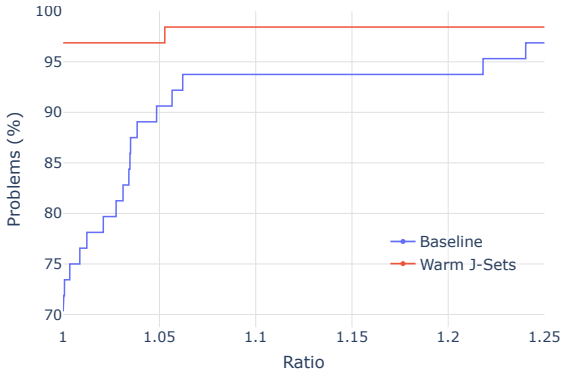
(a) Running time (DS-TS).



(b) Gap (DS-TS).

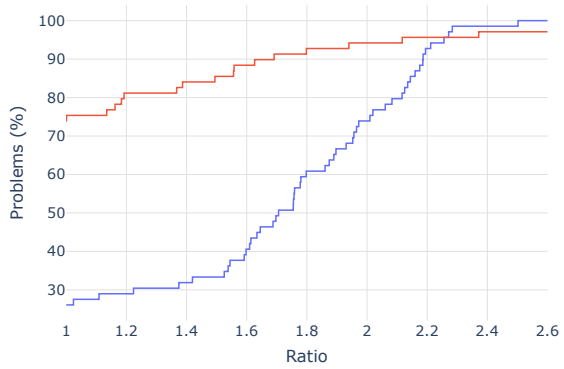


(c) Running time (MINLPLib-TS).

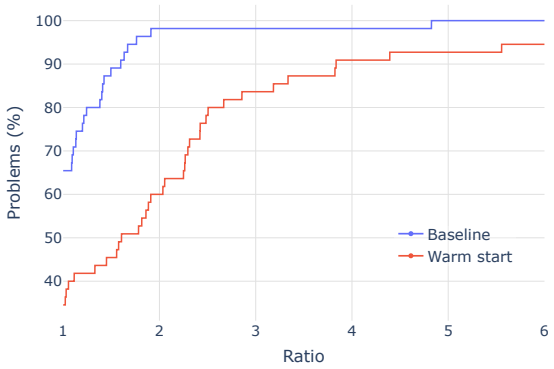


(d) Gap (MINLPLib-TS).

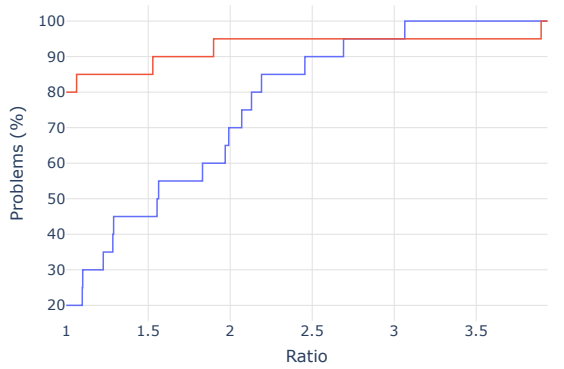
Figure 8: Performance profiles for the warm generation of J-Sets.



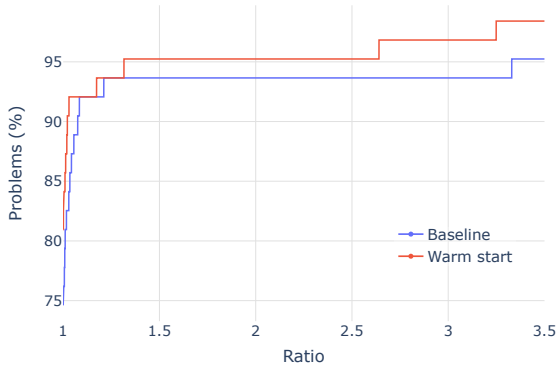
(a) Running time (DS-TS).



(b) Gap (DS-TS).

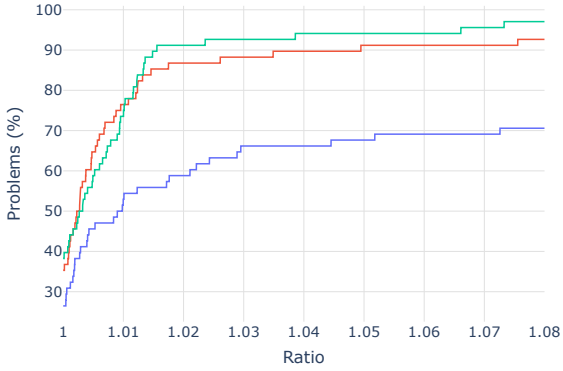


(c) Running time (MINLPLib-TS).

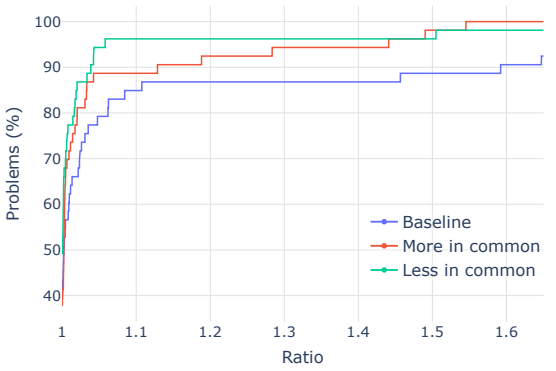


(d) Gap (MINLPLib-TS).

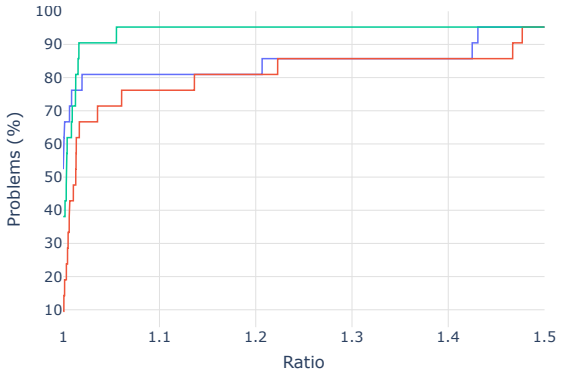
Figure 9: Performance profiles for the warm start on the LP solver.



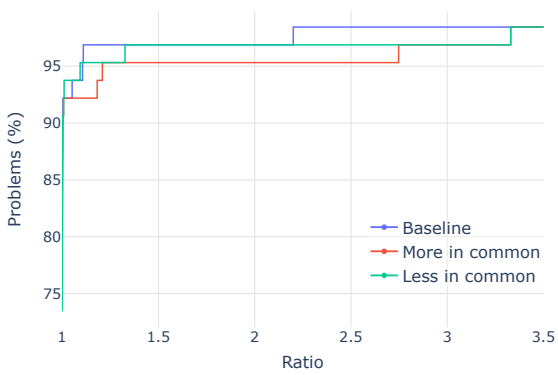
(a) Running time (DS-TS).



(b) Gap (DS-TS).

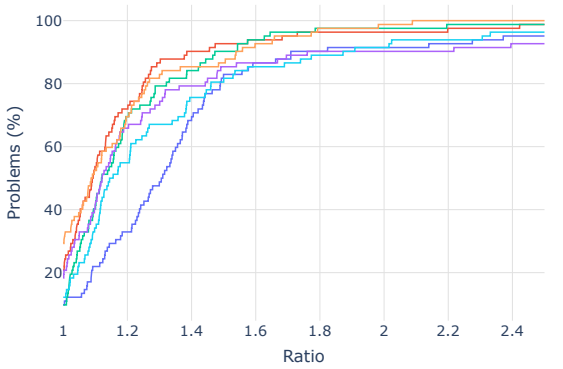


(c) Running time (MINLPLib-TS).

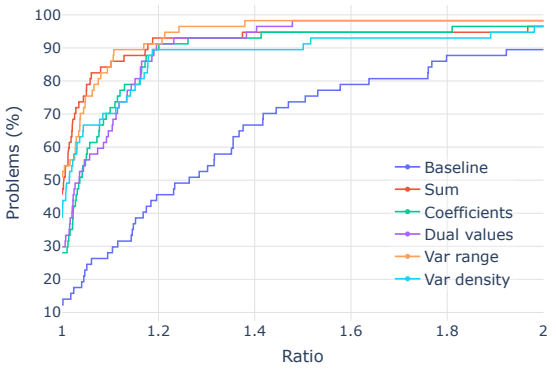


(d) Gap (MINLPLib-TS).

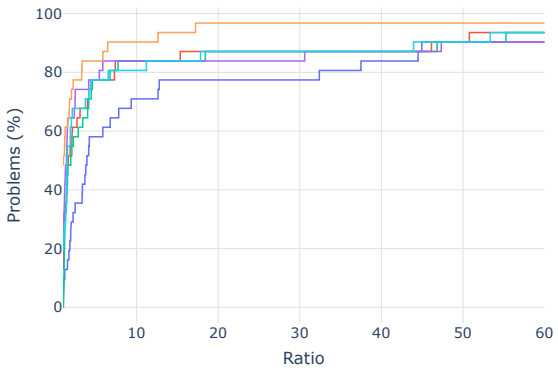
Figure 10: Performance profiles for the products of constraint factors and bound factors.



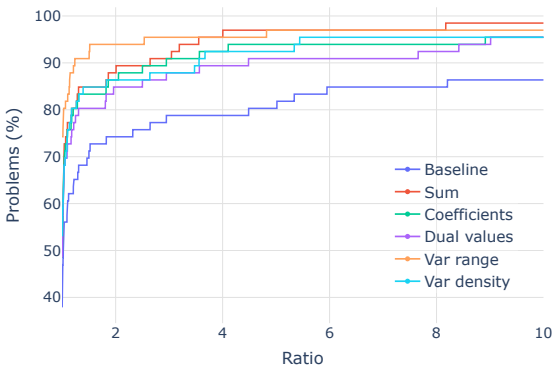
(a) Running time (DS-TS).



(b) Gap (DS-TS).

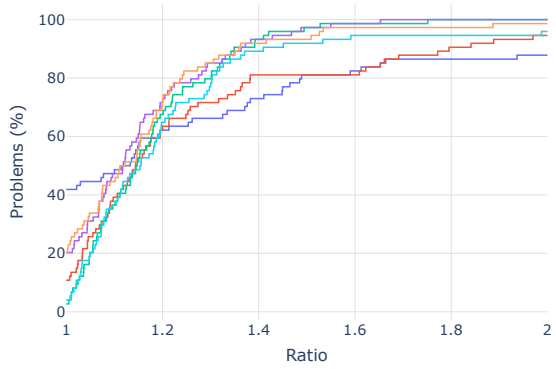


(c) Running time (MINLPLib-TS).

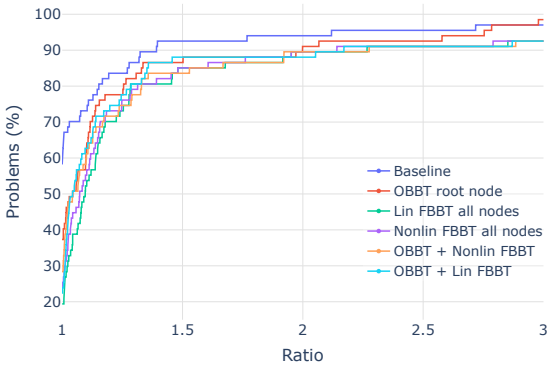


(d) Gap (MINLPLib-TS).

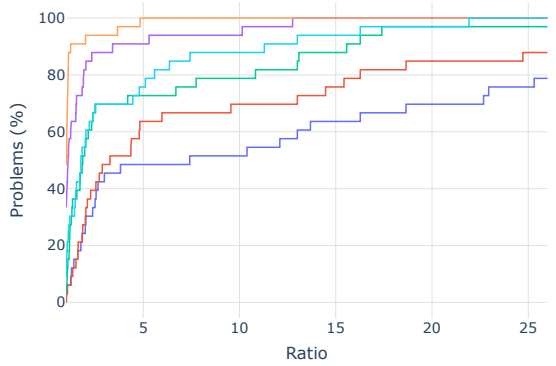
Figure 11: Performance profiles for the different branching criteria.



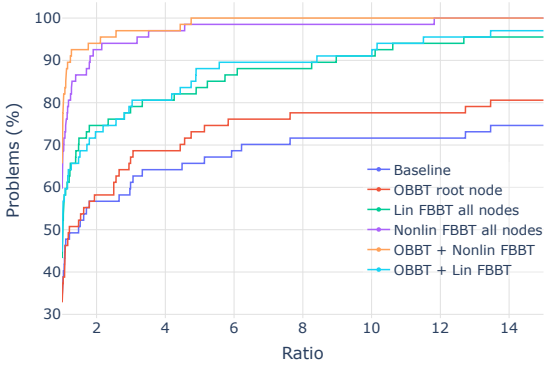
(a) Running time (DS-TS).



(b) Gap (DS-TS).

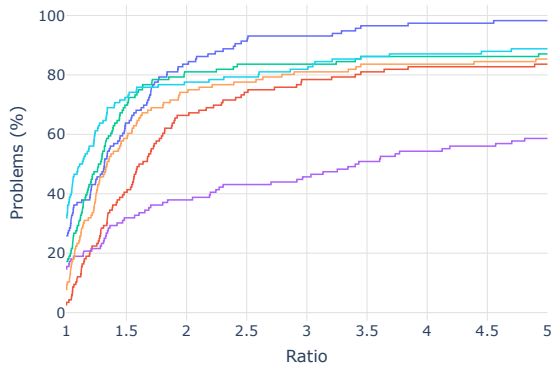


(c) Running time (MINLPLib-TS).

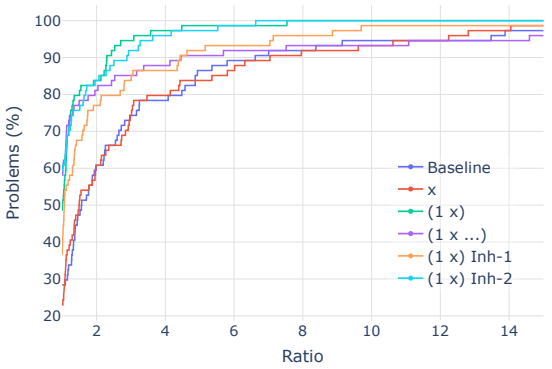


(d) Gap (MINLPLib-TS).

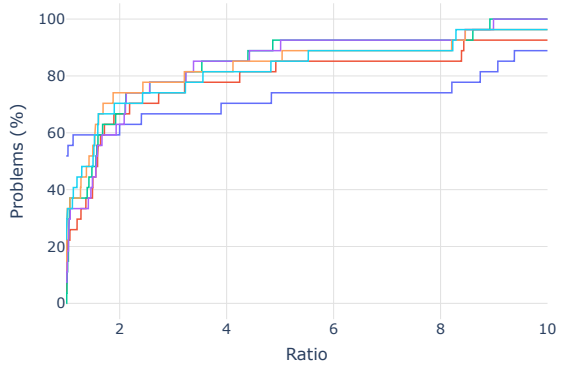
Figure 12: erformance profiles for the bound tightening techniques.



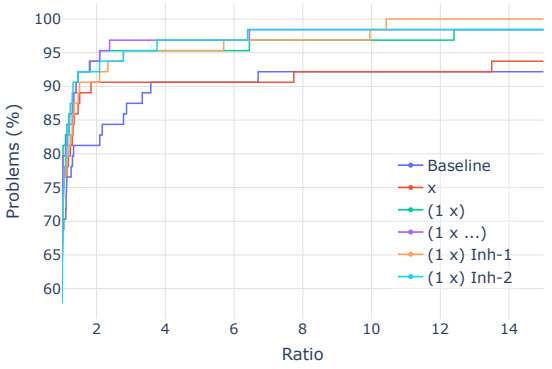
(a) Running time (DS-TS).



(b) Gap (DS-TS).



(c) Running time (MINLPLib-TS).



(d) Gap (MINLPLib-TS).

Figure 13: Performance profiles for the SDP cuts.

B Interaction with RAPOSa

In this appendix, we present the basic steps to run RAPOSa either directly with .nl files or through AMPL. Consider the following polynomial programming problem with box-constrained variables:

$$\begin{array}{ll} \min & x^2 + y^2 + x \\ \text{s.t.} & xy \geq 1 \\ & 1 \leq x \leq 10 \\ & 1 \leq y \leq 8 \end{array}$$

We are going to explain how to solve it through a .nl file. First, we show how to create the .nl file using AMPL, Pyomo or JuMP.

B.1 How to formulate a polynomial programming problem in AMPL and solve it using RAPOSa

In AMPL Fourer et al. (1990), the problem would be written as follows:

```
var x >=1, <= 10;
var y >=0, <= 8;
minimize f: x^2 + y^2 + x;
subject to g: x*y >= 1;
```

If you want to solve this with RAPOSa, you only need to add the following command:

```
option solver raposa;
solve;
```

If you want to run RAPOSa with a specific option, you have to add the following line before the solve command:

```
option raposa_options "outlev=1 maxtime=60";
```

RAPOSa interface with AMPL can be used through NEOS Server on https://neos-server.org/neos/solvers/go:RAPOSa/AMPL.html

B.2 How to formulate a polynomial programming problem in AMPL and convert it to .nl format

In AMPL, the problem would be written as follows:

```
var x >=1, <= 10;
var y >=0, <= 8;
minimize f: x^2 + y^2 + x;
subject to g: x*y >= 1;
```

If you save the above in a file called problem.mod, you only need to run the following command:

```
ampl -ogproblem problem.mod
```

So, you will get the file problem.nl with the problem.

If in addition to the model file problem.mod you have a data file problem.dat, you should run the following command:

```
ampl -ogproblem problem.mod problem.dat
```

If you want to save variables names in problem.col file (.nl format does not store the name of the variables) you need to create a file called options.opt, with the following content:

```
option auxfiles c;
```

and then run the following command:

```
ampl -ogproblem problem.mod options.opt
```

or:

```
ampl -ogproblem problem.mod problem.dat options.opt
```

B.3 How to formulate a polynomial programming problem in Pyomo and convert it to .nl format

In `Pyomo`, the problem would be written as follows:

```
from pyomo.environ import *
m = ConcreteModel()
m.x = Var(bounds=(1,10))
m.y = Var(bounds=(0,8))
def obj(m):
   return (m.x*m.x + m.y*m.y + m.x)
def cons(m):
   return (m.x * m.y >= 1)
m.obj = Objective(rule=obj)
m.cons = Constraint(rule=cons)
```

and in order to write it in `.nl` format, it is necessary to add `write` command at the end, as shown below:

```
from pyomo.environ import *
m = ConcreteModel()
m.x = Var(bounds=(1,10))
m.y = Var(bounds=(0,8))
def obj(m):
   return (m.x*m.x + m.y*m.y + m.x)
def cons(m):
   return (m.x * m.y >= 1)
m.obj = Objective(rule=obj)
m.cons = Constraint(rule=cons)
m.write("problem.nl")
```

Thus, running the previous file in the python console, you obtain a `.nl` file which is called `problem.nl`.

If you want to save variables names in `problem.col` file (`.nl` format does not store the name of the variables) you need to include the option `symbolic_solver_labels`, as shown below:

```
from pyomo.environ import *
m = ConcreteModel()
m.x = Var(bounds=(1,10))
m.y = Var(bounds=(0,8))
def obj(m):
   return (m.x*m.x + m.y*m.y + m.x)
def cons(m):
   return (m.x * m.y >= 1)
m.obj = Objective(rule=obj)
m.cons = Constraint(rule=cons)
m.write("problem.nl",io_options="symbolic_solver_labels":True})
```

B.4 How to formulate a polynomial programming problem in JuMP and convert it to .nl format

In `JuMP`, the problem would be written as follows:

```
using JuMP, AmplNLWriter
m = Model(solver=AmplNLSolver("bonmin"))
@variable(m, 1 <= x <= 10 )
@variable(m, 0 <= y <= 8 )
@NLobjective(m, Min, x^2 + y^2 + x )
@NLconstraint(m, x*y >= 1.0 )
```

and in order to write it in `.nl` format, it is necessary to add several lines, as shown below:

```
using JuMP, AmplNLWriter
m = Model(solver=AmplNLSolver("bonmin"))
@variable(m, 1 <= x <= 10 )
@variable(m, 0 <= y <= 8 )
@NLobjective(m, Min, x^2 + y^2 + x )
@NLconstraint(m, x*y >= 1.0 )
JuMP.build(m)
m2 = m.internalModel.inner
AmplNLWriter.make_var_index!(m2)
AmplNLWriter.make_con_index!(m2)
f = open("./problem.nl","w")
AmplNLWriter.write_nl_file(f,m2)
close(f)
```

Thus, running the previous file in the `Julia` console, you obtain a `.nl` file which is called `problem.nl`.

If you want to save variables names in `problem.col` file (`.nl` format does not store the name of the variables) you need to create the `problem.col` file manually. In this case, the content would be as follows:

```
x
y
```

B.5 How to solve a problem in .nl format with `RAPOSa`

Solve a problem in `.nl` format with `RAPOSa` is easy. You only need to run the following command:

```
./raposa problem.nl
```

and `RAPOSa` will be executed with default options.

If you want to run it with specific options, for example a time limit of 30 seconds and `Gurobi` as linear solver, you should run the following command:

```
./raposa problem.nl -maxtime 30 -linsolver gurobi
```

If you want to know all available options, you should run the following command:

```
./raposa -help
```

In this case, if you run the command

```
./raposa problem.nl -outlev 1 -output output.json
```

you will obtain the following output:

```
================================================================================
```

```
 RAPOSa v1.1.2 (2020.04.21)
 Copyright (C) 2019 ITMATI - Univ. de Santiago de Compostela (USC). All rights
     reserved.
 This software is free for non-commercial purposes.
 Full licence information can be found in the LICENSE.txt file.
 http://www.itmati.com/RAPOSa/index.html
================================================================================
 Linear solver: googleor-glop
 Nonlinear solver: ipopt
================================================================================
  Iteration      Time (s)      Lower Bound      Upper Bound      Relative Gap
      Absolute Gap     Feas error
         1          0.01           2.0000           3.0000        0.33322226
             1.0000      0.00000000
         4          0.02           3.0000           3.0000        0.00000000
             0.0000      0.00000000

Global solution found after 4 iterations and 0.023525 seconds.
Objective: 3
```

and the following `output.json` file:

```
{
    "Total time": 0.0234821,
    "Number of iterations": 4,
    "Upper bound": 3,
    "Lower bound": 3,
    "Absolute gap": 5.07746e-09,
    "Relative gap": 1.69192e-09,
    "Feasilibity error": 0,
    "Solution": {
           "x[0]" : 1,
           "x[1]" : 1
    }
}
```