Solving Multiplicative Programs by Binary-encoding the Multiplication Operation

Payman Ghasemi Saghand
Department of Industrial and Management Systems Engineering, University of South Florida, Tampa, FL 33620, USA
payman@usf.edu

Fabian Rigterink
ExxonMobil Upstream Integrated Solutions, Spring, TX 77389, USA fabian.rigterink@exxonmobil.com

Vahid Mahmoodian and Hadi Charkhgard
Department of Industrial and Management Systems Engineering, University of South Florida, Tampa, FL 33620, USA
mahmoodian@usf.edu (V. Mahmmodian) and hcharkhgard@usf.edu (H. Charkhgard)

Multiplicative programs in the form of maximization and/or minimization have numerous applications in conservation planning, game theory, and multi-objective optimization settings. In practice, multiplicative programs are challenging to solve because of their multiplicative objective function (a product of continuous or integer variables). These challenges are twofold: 1. As the number of factors in the objective increases, so does the solution time, and the problems become computationally expensive to solve. 2. If all factors are in $(0, 1)$ or in $(1, \infty)$, the objective may cause ill-conditioning and numerical instability. The solution methods proposed in this paper help overcome both of these challenges. The main idea is to binary-encode the multiplication operation analogously to how a computer conducts it internally. This not only solves the aforementioned numerical issues but also allows us to develop a new family of solution methods for multiplicative programs. One such method is to solve the multiplicative programs bit-by-bit, i.e., iteratively computing the optimal value of each bit of the objective function. In an extensive computational study, we explore a number of solution methods that solve multiplicative programs faster and more accurately.

Key words: Multiplicative program; Binary-encoding; Multi-linear optimization, Mixed integer second order cone programming, Nonconvex nonlinear programming

1. Introduction

The focus of this paper is on the following class of optimization problems, known as (linear) *multiplicative programs*,

$$\max \text{ or } \min \left\{ \prod_{i\in I} y_i(\boldsymbol{x}) : \; \boldsymbol{x} \in \mathcal{X}, \; \boldsymbol{y}(\boldsymbol{x}) \geq \boldsymbol{0} \right\}, \tag{1}$$

where $I := \{1, \dots, p\}$, $\mathcal{X} \subseteq \mathbb{R}^{n_C} \times \mathbb{Z}^{n_I}$ ($n_C, n_I \geq 0$ being the number of continuous and integer variables, respectively) is the set of feasible solutions described by only linear constraints, and $\boldsymbol{y}(\boldsymbol{x})$ is a vector of p non-negative linear functions of $\boldsymbol{x} \in \mathcal{X}$. A multiplicative program is referred to as a *Maximum Multiplicative Program (MMP)* if it is in the form of maximization. Similarly, it is referred to as a *minimum Multiplicative Program (mMP)* if it is in the form of minimization. This study is motivated by two observations.

The first one is that MMPs and mMPs are typically studied independently in the existing body of literature as they are completely different in nature. This is highlighted by the fact that when there are no integer variables, i.e., $n_I = 0$, MMPs can be solved in polynomial time while mMPs remain NP-hard (Charkhgard et al. 2018, Shao and Ehrgott 2014, 2016). Consequently, not surprisingly, more effective solution methods exist for MMPs that cannot be used for solving mMPs (Saghand et al. 2019). One such method is to replace the multiplicative objective function by its corresponding log-transformed function, i.e., $\sum_{i\in I} \log(y_i(\boldsymbol{x}))$, and to solve the transformed problem by a mixed integer convex programming solver. Another method, which in practice is even faster, first replaces the multiplicative objective function by its corresponding geometric mean function, i.e., $\sqrt[p]{\prod_{i\in I} y_i(\boldsymbol{x})}$, and then transforms the problem into a mixed integer Second Order Cone Program (SOCP) in order to be solved by powerful commercial solvers such as CPLEX and Gurobi (Charkhgard et al. 2018). Overall, there is a lack of custom-built exact solution methods that can be used for both mMPs and MMPs, and this study attempts to address this gap.

The second observation is that for large values of p, the optimal objective value of Problem (1) tends to be either very large or very small due to the multiplicative nature of its objective function. We refer to this issue as *the curse of multiplication*. For example, suppose that $p = 100$ and

$y_1 = \cdots = y_{100} = 99$ in an optimal solution. The optimal objective value, 99^{100}, is an astronomical number that far exceeds the capabilities of any state-of-the-art method to handle it. Moreover, for such magnitudes of objective values, the (typical) *relative* optimality gap tolerance that solvers use, e.g., 10^{-4}, is insufficient as the *absolute* optimality error continues to be beyond our imagination. Therefore, existing solvers need to select smaller values for their relative optimality gap tolerance, however, that can decrease their performance significantly and possibly create other forms of numerical issues. It is worth mentioning that transformation-based solution methods (e.g., the log-transformation) can possibly resolve the curse of multiplication for some multiplicative programs. However, transformations themselves can create other forms of numerical issues because of their nonlinear nature and the fact that their outcomes can be irrational numbers. For example, $\log_{10} 99 = 2\log_{10} 3 + \log_{10} 11$ is obviously irrational. Overall, there is a lack of custom-built exact solution methods that can truly resolve the curse of multiplication, and this study attempts to address this gap.

1.1. Applications

An important application of multiplicative programs is in the field of conservation planning. This field deals with the issues related to preserving and/or increasing biodiversity (Beyer et al. 2016). Preserving biodiversity is crucial to human societies and the future of planet Earth, and its slow erosion constitutes a threat as consequential as that posed by climate change (Billionnet 2013). Preserving biodiversity is a vague goal and needs to be translated into measurable and definable objectives. Unfortunately, such a translation is not a trivial task and is known to be one of the significant challenges in environmental management (Failing and Gregory 2003, Nicholson and Possingham 2006). Therefore, a common approach in the literature of conservation planning is to compute extinction risks and use them for measuring biodiversity and comparing different solutions in conservation planning problems (Nicholson and Possingham 2006). Specifically, the existing body of literature suggests that conservation planning problems need to be formulated and solved as multiplicative programs.

In such multiplicative programs, $\mathcal{X}$ represents the set of feasible solutions of a conservation planning problem. A feasible solution can represent, for example, which parcels of land can be purchased for protection, i.e., creating a nature reserve, based on the available budget as well as spatial and temporal constraints. Moreover, p represents the number of important/endangered species under consideration. Note that p can be a large number in practice as natural resource managers may be interested in considering dozens of species for a conservation planning problem (Sierra-Altamiranda et al. 2020). For each species $i \in I$, let $f_i(\boldsymbol{x})$ be a (linear) function that captures the extinction risk corresponding to solution $\boldsymbol{x} \in \mathcal{X}$. Based on these notations, the literature suggests that mMPs can be solved if managers are interested in minimizing the probability of more species being extinct. For mMPs, we should set $y_i(\boldsymbol{x}) = f_i(\boldsymbol{x})$ for each species $i \in I$. The literature also suggests that MMPs can be solved if managers are interested in minimizing the probability of all species being extinct. Then, for MMPs, we should set $y_i(\boldsymbol{x}) = 1 - f_i(\boldsymbol{x})$ for each species $i \in I$.

In addition to conservation planning, multiplicative programs have applications in many other domains. For example, mMPs have applications in bond portfolio management, economic analysis, and VLSI chip design (Benson and Boger 2000). MMPs, on the other hand, can be used to compute the Nash solution in bargaining problems in the field of cooperative game theory (Nash Jr 1950). In bargaining problems, the multiplicative function is sometimes referred to as the *Nash Social Welfare function* (Caragiannis et al. 2019). Hence, in bargaining problems, MMPs can be viewed as maximizing the Nash Social Welfare function over the feasible set of actions. Note that many real-world bargaining problems require integer decision variables, e.g., the allocation of indivisible goods among heirs (Caragiannis et al. 2019). MMPs can also be used in computing an equilibrium for Fisher's linear or Kelly's capacity allocation market (Vazirani 2012) or systems reliability problems (Abouei Ardakan et al. 2016, Feizabadi and Jahromi 2017, Zhang and Chen 2016).

1.2. Contributions

The main contribution of our research is to develop techniques in order to binary-encode the multiplication operation in the objective function of a multiplicative program. Using the proposed techniques, a family of novel solution methods will be introduced for solving both MMPs and mMPs. The new family of solution methods can resolve the curse of multiplication as they do not necessarily need to deal with the magnitude of the objective values. In fact, they are designed to compute the value of bits of the optimal objective value from the most significant bit to the least significant one. Overall, a nice property of the proposed family of solution methods is that they only solve a finite number of integer linear programs to compute an optimal solution for a multiplicative program. Depending on a specific solution method, the number of integer linear programs that need to be solved is at least one and at most equal to the number of bits required to represent the objective of a multiplicative program. Solution methods in our proposed family are different mainly because of two reasons.

The first reason is that the binary-encoding of the multiplication operation can be done in multiple ways (if $p > 2$) which in turn results in different equivalent reformulations. In this study, we explore two categories of reformulations which we will refer to as *Altogether* and *Nested*. The Altogether strategy generates a reformulation that directly binary-encodes $\prod_{i \in I} y_i$ as a whole. To run the Nested strategy, first, a sequence will be generated, e.g., $(1, 2, 3, \ldots, p)$, reflecting how the multiplication process should be carried out. Then, in the reformulation, $p - 1$ binary-encoded multiplications of two variables will be generated based on the sequence. For example, for the sequence $(1, 2, 3, \ldots, p)$, the reformulation contains the binary-encoded multiplication $y_1 \times y_2$, the binary-encoded multiplication $y' \times y_3$ (where y' is the outcome of $y_1 \times y_2$), and so on.

The second reason is that after creating a reformulation, different search mechanisms can be developed to solve it. In this study, we consider two main categories of search mechanisms which we will refer to as *One-shot* and *Bitwise*. One-shot means that the reformulation should be solved directly by solving a mixed integer linear program. In this strategy, the optimal values of all bits will be obtained at the same time. For the Bitwise reformulation, the values of bits will be obtained iteratively, i.e., exploring one bit at a time. To compute the optimal value of each bit, one mixed integer linear program must be solved (if needed). We also develop two cut-generating techniques

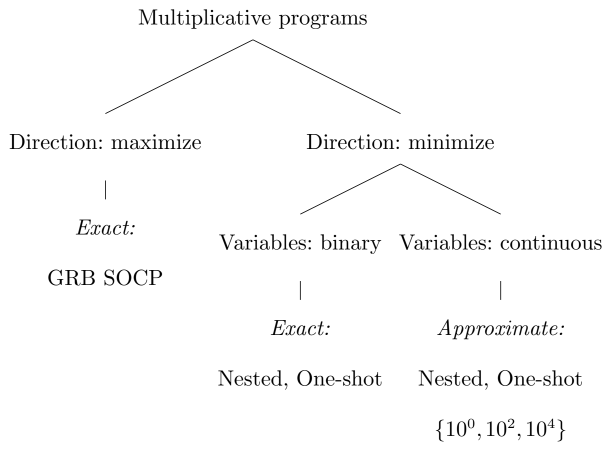


Figure 1 High-level summary of computational results as a tree (recommended solution approaches)

that can be combined with the Bitwise strategy to tighten the reformulations in each iteration and possibly reduce the number of mixed integer linear programs needed.

The Bitwise strategy is naturally slower as it is the only strategy that fully resolves the curse of multiplication. The One-shot strategy, on the other hand, is faster as it does not resolve the curse of multiplication but directly solves an integer linear programming reformulation of a multiplicative program. In this study, we first provide some numerical examples to show that the Bitwise strategy is the best strategy to deal with large values of p as it does not run into any numerical instabilities. We then conduct a detailed computational study on instances with $p \in \{2,3,4\}$ and compare the One-shot strategy with GRB SOCP (Gurobi mixed integer SOCP solver) on MMPs and GRB Nonconvex (Gurobi mixed integer nonconvex quadratic solver) on mMPs. The computational study provides a practical guide indicating which solution method to use for which problem variation. A high-level summary of this guide is shown in Figure 1, which is a tree representation of the results later shown in Section 5, Table 5. In this tree, for a given problem variation, we list the solution methods with the shortest solution times. Since the aforementioned applications of multiplicative programs are typically either pure binary or pure continuous cases, we employ two classes of (random) instances in our study. We note that to binary-encode the multiplication operation, $y_i(\boldsymbol{x})$ should only take integer values for all $i \in I$. Otherwise, our proposed family of solution methods are only approximate methods. For such instances, the function $y_i(\boldsymbol{x})$ will be assumed to be integer for all $i \in I$ after being multiplied by a sufficiently large number. The sufficiently large number should be determined based on the level of precision required. In our computational study, we consider three possible scenarios for the value of the multiplier: 10^0, 10^2, and 10^4. Overall, our results show that for MMPs, GRB SOCP is the best choice. For mMPs, however, our proposed family of solution methods beat the off-the-shelf benchmark most of the time.

1.3. Outline

The remainder of this paper has the following outline: in Section 2, we introduce the methodology at a high level, and in Section 3, we introduce the methodology in detail. In Section 4, we show the power of the Bitwise search mechanism in resolving the curse of multiplication for some extreme examples. In Section 5, we compare the solution methods with the aforementioned benchmarks in an extensive computational study that we ran on 900 instances spanning a total of 5,400 runs grouped in 4 experiments, with each experiment designed to find the best solution method for a given problem variation. Finally, in Section 6, we conclude the paper with some final recommendations and remarks.

2. High-level methodology

A multiplicative program can be stated as

$$\max_{\boldsymbol{x},\boldsymbol{y}} \text{ or } \min \quad \prod_{i \in I} y_i \tag{2}$$

$$\text{s.t.} \quad \boldsymbol{y} = D\boldsymbol{x} + \boldsymbol{d}, \tag{3}$$

$$A\boldsymbol{x} \leq \boldsymbol{b}, \tag{4}$$

$$\boldsymbol{x}, \boldsymbol{y} \geq \boldsymbol{0}, \ \boldsymbol{x} \in \mathbb{R}^{n_C \times n_I}, \ \boldsymbol{y} \in \mathbb{R}^{p_C \times p_I}, \tag{5}$$

where n_C and n_I denote the number of continuous and integer decision variables in the space of x-variables, respectively. Moreover, p_C and p_I denote the number of continuous and integer decision variables in the space of y-variables, respectively. Given $n := n_C + n_I$ and $p := p_C + p_I$, D is a

$p \times n$ matrix representing the coefficients of the multiplicative terms. Similarly, $\boldsymbol{d}$ is a vector of size p representing the constants of the multiplicative terms. Further, A is an $m \times n$ matrix that denotes the technological coefficients, with m being the number of linear constraints. Finally, $\boldsymbol{b}$ represents the m-sized vector of right-hand side values. A powerful off-the-shelf solver that can solve any multiplicative program is Gurobi. For an MMP, users need to transform the problem into a (mixed integer) SOCP as described in Ben-Tal and Nemirovski (2001), Charkhgard et al. (2018) and then call GRB SOCP (Gurobi mixed integer SOCP solver) to solve it. To solve mMPs, users can use GRB Nonconvex (Gurobi mixed integer nonconvex quadratic solver). Obviously, for $p > 2$, an mMP is not quadratic. However, it can be brought into quadratic form by introducing some auxiliary variables. For example, instead of $y_1y_2y_3$, one can write $y'y_3$ where $y' = y_1y_2$.

The breakthrough idea of the proposed research is to binary-encode the multiplication operation (analogously to how a computer conducts it internally) such that the objective value of any solution is represented by bits. In that case rather than dealing with the entire optimal objective value, computing the bits of the optimal objective value becomes crucial. Each bit is a binary decision variable and the values of the bits can be determined in an iterative fashion from the most significant bit to the least significant one. In each iteration, the value of one bit of the optimal objective value will be computed and then its value will be fixed for future iterations. Describing the full mathematical model of a binary-encoded reformulation of the multiplicative program requires heavy notation. Instead of introducing the full notation now (we will do so later), in this section, we present the key idea using a simple example with $p_I = 2$ and $p_C = 0$. Suppose that it is known that $y_1, y_2 \leq 10$. To develop a binary-encoded reformulation, the following steps need to be taken:

- *Step 1:* Find an upper bound for the optimal objective value. Observe that, by defining $z = y_1y_2$, we have $z \leq 100$. This implies that the binary representation of the value of the product has at most $\lfloor \log_2 100 \rfloor + 1 = 7$ bits. Let $\hat{z}_0, \dots, \hat{z}_6$ denote the values of the bits to binary-encode z where $\hat{z}_0$ is the least significant bit and $\hat{z}_6$ is the most significant one. The optimal values of $\hat{z}_0, \dots, \hat{z}_6$ are unknown and the idea is to compute them one by one. As an example, an illustration of how to compute $\hat{z}_5$ is shown in Figure 2 where we binary-encode the multiplication of 14 and 5.

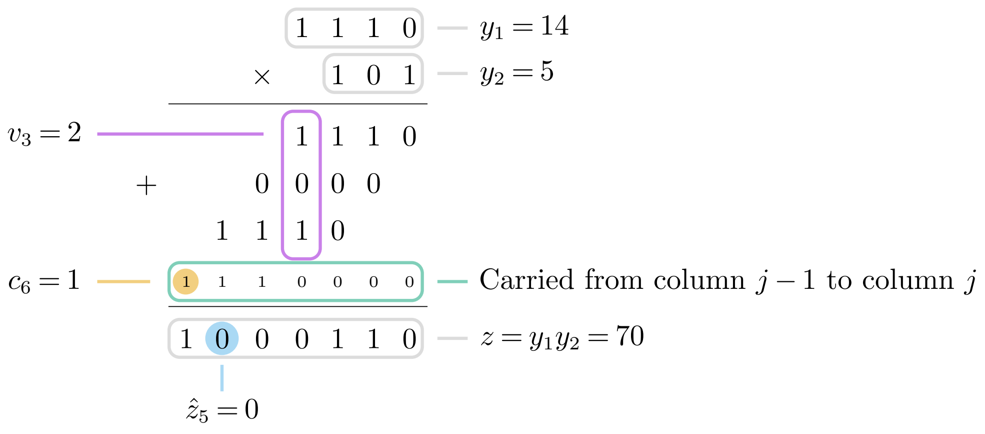


Figure 2 Binary multiplication of two binary-encoded numbers (numbering/indexing starts from zero and is done from right to left, e.g., column $j-1$ is to the right of column j and $\hat{z}_4$ is to the right of $\hat{z}_5$)

- *Step 2:* Represent y_1 and y_2 in a binary-encoded format. This can be done easily (using linear equations) when it is known that both y_1 and y_2 can only take integer values:

$$y_1 = \sum_{i=0}^{\lfloor \log_2 10 \rfloor + 1} 2^i \hat{y}_{1,i} \quad \text{and} \quad y_2 = \sum_{i=0}^{\lfloor \log_2 10 \rfloor + 1} 2^i \hat{y}_{2,i} \tag{6}$$

where $\hat{y}_{1,i}$ and $\hat{y}_{2,i}$ capture the value of the bit with index number i for y_1 and y_2, respectively. Note that $\lfloor \log_2 10 \rfloor + 1 = 4$, and hence, only 4 bits each are required to represent y_1 and y_2.

- *Step 3:* Apply the principles of the *binary multiplication* (see Figure 2) for capturing the value of $\hat{z}_j$ for all $j \in \{0, 1, \ldots, 6\}$. As an aside, we note that the 'binary multiplication' is basically the *Fast Fourier Transform (FFT) based multiplication of integers* (Borwein and Bailey 2008), which is closely related to a multiplication method commonly referred to as *ancient Egyptian multiplication* or *Ethiopian/Russian multiplication*[1].

  – The sum of all binary products of column j is denoted by v_j and it can be captured by the following equation for all $j \in \{0, \ldots, 6\}$:

$$v_j = \sum_{\substack{i,i' \in \{0,1,2,3\}: \\ i+i'=j}} \hat{y}_{1,i}\ \hat{y}_{2,i'}. \tag{7}$$

[1] As the name suggests, such multiplication methods were already known to the Ancient Egyptians, as can be seen e.g. in *The Rhind Mathematical Papyrus* dating to around 1550 BC (see the *British Museum* objects EA10057 and EA10058).

– The carried value from column $j-1$ to j is denoted by c_j and it can be captured by the following equation for all $j \in \{0,\ldots,6\}$:

$$c_j = \begin{cases} \left\lfloor \dfrac{v_{j-1}+c_{j-1}}{2} \right\rfloor & \text{if } j \neq 0 \\ 0 & \text{if } j = 0 \end{cases}. \tag{8}$$

– Finally, the value of $\hat{z}_j$ can be captured by the following equation for all $j \in \{0,\ldots,6\}$:

$$\hat{z}_j = \begin{cases} v_j + c_j - 2c_{j+1} & \text{if } j \neq 6 \\ v_j + c_j & \text{if } j = 6 \end{cases}. \tag{9}$$

An illustration of how to compute v_3, c_6, and $\hat{z}_5$ is shown in Figure 2. In light of the above, the binary-encoded reformulation of the multiplicative programs for this example can be stated as

$$\begin{aligned} \max_{\boldsymbol{x},\boldsymbol{y},\hat{\boldsymbol{z}},\hat{\boldsymbol{y}},\boldsymbol{c},\boldsymbol{v}} \text{ or } \min \quad & \sum_{j=0}^{6} 2^j \hat{z}_j \\ \text{s.t.} \quad & (3)\text{–}(5) \text{ and } (6)\text{–}(9), \\ & \hat{y}_{1,i}, \hat{y}_{2,i} \in \{0,1\}, \qquad i \in \{0,1,\ldots,3\}, \\ & \hat{z}_j \in \{0,1\}, \text{ and } c_j, v_j \in \mathbb{N}_0 \quad j \in \{0,1,\ldots,6\}, \end{aligned} \tag{10}$$

where $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$. Note that, in Problem (10), the variable $\hat{z}_j$ represents the bit $j \in \{0,1,\ldots,6\}$ of the objective value and it naturally takes a binary value. Observe that in Problem (10), only Constraint (7) is nonlinear. However, this constraint can be easily linearized. In Constraint (7), the term $\hat{y}_{1,i}\hat{y}_{2,i'}$ is nonlinear but $\hat{y}_{1,i}$ and $\hat{y}_{2,i'}$ are binaries. Hence, the term can be replaced by a new binary variable $u_{i,i'}$ and by adding the following three inequalities (often referred to as *McCormick envelopes* (Dey and Gupte 2015, Luedtke et al. 2012)):

$$u_{i,i'} \leq \hat{y}_{1,i}, \qquad u_{i,i'} \leq \hat{y}_{2,i'}, \qquad u_{i,i'} \geq \hat{y}_{1,i} + \hat{y}_{2,i'} - 1.$$

Note that it is not necessary to impose the integrality condition on $u_{i,i'}$ as it naturally takes integer values. Also, note that Constraint (8) appears nonlinear but can be written in linear form without introducing any new variables as follows:

$$c_0 = 0,$$

$$2c_j \geq v_{j-1} + c_{j-1} - 1, \qquad j \in \{1, \ldots, 6\},$$

$$2c_j \leq v_{j-1} + c_{j-1}, \qquad j \in \{1, \ldots, 6\}.$$

Overall, the proposed reformulation is a mixed integer linear program that can be solved directly by commercial solvers such as CPLEX and Gurobi. By directly solving the proposed reformulation, the optimal values of all bits will be determined at the same time. However, solving the proposed binary reformulation directly does not resolve the curse of multiplication as the magnitude of the objective value of the reformulation remains exactly the same as in the original formulation. Observe now that it is not necessary to solve the binary-encoded reformulation directly. Instead, the values of the bits of the optimal objective value can be obtained in an iterative fashion from the most significant bit to the least significant one. More formally, at iteration $t = 0, \ldots, 6$, Problem (11) needs to be solved:

$$\begin{aligned}
\max_{\boldsymbol{x},\boldsymbol{y},\hat{\boldsymbol{z}},\hat{\boldsymbol{y}},\boldsymbol{c},\boldsymbol{v}} \text{ or } \min \quad & \hat{z}_{6-t} \\
\text{s.t.} \quad & \hat{z}_{6-t'} = \hat{z}^*_{6-t'}, && t' \in \{0, \ldots, t-1\}, \\
& (3)\text{–}(5) \text{ and } (6)\text{–}(9), \\
& \hat{y}_{1,i}, \hat{y}_{2,i} \in \{0,1\}, && i \in \{0, 1, \ldots, 3\}, \\
& \hat{z}_j \in \{0,1\}, \text{ and } c_j, v_j \in \mathbb{N}_0, && j \in \{0, 1, \ldots, 6\},
\end{aligned} \tag{11}$$

Note that in Problem (11), $\hat{z}^*_{6-t'}$ represents the optimal value of Problem (11) obtained in iteration t'. In other words, after each iteration, the value of its corresponding bit will be fixed for future iterations. Observe that the proposed iterative method completely resolves the curse of multiplication. Moreover, users can impose any desired level of precision as the termination condition on the proposed algorithm. The level of precision then defines how many bits need to be accurately computed. Overall, in the proposed iterative method only a mixed integer linear program needs to be solved in each iteration whose objective value is either zero or one (assuming the original formulation is feasible). In Section 3.2.2, we discuss that it is not necessary to solve *all* mixed integer linear programs as the optimal value of some bits can be determined directly based on the information obtained in the previous iterations.

In summary, from the discussions above, after developing a binary-encoded reformulation for a multiplicative program, two search mechanisms can be used to solve it: the direct approach, One-shot, and the iterative one, Bitwise. We note that in the simple example above, we assumed that $p_C = 0$. However, this is not an impractical assumption since if $p_C > 0$, it can still be approximated to any desired level of precision by a multiplicative program with $p_C = 0$. Specifically, we can assume that only a certain number of digits after the decimal point is needed/necessary for fractional y-variables. If, for example, two digits after the decimal point are needed for y_i, then we can still assume that y_i is an integer variable after multiplying the right-hand-side of its associated constraint in (3) by 100. Such transformations are of course costly as they will increase the maximum number of bits required to represent the optimal objective value of a multiplicative program. However, they can be used to solve multiplicative programs involving fractional y-variables. We note that in terms of implementation, there are a few points that should be considered when dealing with continuous variables. We will address these in Section 3.4.

Finally, we note that while for instances with $p = 2$, there is only one way to develop a binary-encoded reformulation, for $p > 2$ there are many possible ways. In this study, we focus only on two main categories of reformulations: Nested and Altogether. For example, if $p = 3$, a Nested reformulation is $(y_1 y_2) y_3$ and the Altogether reformulation is $(y_1 y_2 y_3)$. Such binary-encoded reformulations are substantially different in terms of the number of variables and constraints that they introduce. For example, for $p = 3$, the Altogether reformulation comes with the disadvantage of making the linearization of Equation (7) more difficult because instead of bilinear terms, trilinear terms will appear in Equation (7). However, it also comes with the advantage of having only minor impacts on increasing the complexity of Equations (6), (8), and (9). The Nested reformulation, on the other hand, has exactly the opposite effect. Later, in Section 3.1.2, we show that a nice property of the Nested reformulation is that it keeps the size of the problem polynomially bounded while the Altogether reformulation does not guarantee that.

Table 1 Frequently used notations (subscripts and superscripts will be added whenever needed)

Variables	
z	An integer variable used for representing the product of some y variables
$\hat{z}$	A binary variable used for representing a bit of the binary-encoded version of a z variable
$\hat{y}$	A binary variable used for representing a bit of the binary-encoded version of a y variable
v	An integer variable used for representing the summation of a column in the process of binary multiplication
c	An integer variable used for representing the carried value from a column to its next column in the process of binary multiplication
u	A binary variable used for representing McCormick envelopes of the product of some $\hat{y}$ variables
Parameters	
n^z	An integer parameter showing the maximum number of bits required to represent a z variable
n^y	An integer parameter showing the maximum number of bits required to represent a y variable

3. Detailed methodology

In this section, we provide the detailed description of our proposed family of solution methods for multiplicative programs. Our notational convention in this section is summarized in Table 1. We start this section by assuming $p_C = 0$ for now, but we will later relax this condition.

3.1. Binary-encoded reformulations

3.1.1. Products of length two: the *Nested* reformulation We are now going to implement the FFT-based multiplication as constraints to reformulate a multiplicative program. For this and for the following section, let

- $\overline{y}_i$ denote the upper bounds on y_i, $i \in I$ (calculated in preprocessing),
- $n_i^y = \lfloor \log_2(\overline{y}_i) \rfloor + 1$ denote the number of binary variables needed to encode y_i, $i \in I$,
- $J_i^y = \{0, \dots, n_i^y - 1\}$, $i \in I$,
- $\hat{y}_{i,j} \in \{0,1\}$, $j \in J_i^y$, denote the binary variables to encode y_i, $i \in I$.

In this formulation, the idea is to interleave (or *nest*) the product (2) into products of length two – that is – for all $i \in I$, we set

$$z_i = y_i \times \begin{cases} 1, & i = 1, \\ z_{i-1}, & i \in I \setminus \{1\}, \end{cases}$$

where $(2) = \prod_{i \in I} y_i = z_p$. Let

- $\overline{z}_i = \prod_{\substack{i' \in I: \\ i' \leq i}} \overline{y}_{i'}$ denote the upper bounds on z_i, $i \in I$,
- $n_i^z = \lfloor \log_2(\overline{z}_i) \rfloor + 1$ denote the number of binary variables needed to encode z_i, $i \in I$,

- $J_i^z = \{0,\dots,n_i^z-1\}$, $i \in I$,
- $\hat{z}_{i,j} \in \{0,1\}$, $j \in J_i^z$, denote the binary variables to encode z_i, $i \in I$,
- $u_{i,j,j'} \in [0,1]$, $i \in I\setminus\{1\}$, $j \in J_i^y$, $j' \in J_{i-1}^z$, be a continuous variable that is naturally binary and that encodes the $\hat{y}\hat{z}$ products,
- $v_{i,j} \in \mathbb{N}_0$, $i \in I\setminus\{1\}$, $j \in J_i^z$, be an integer variable that represents the j-th column sum of the u variables,
- $c_{i,j} \in \mathbb{N}_0$, $i \in I\setminus\{1\}$, $j \in J_i^z$, be an integer variable that represents the carried value from column $j-1$ to column j.

We reformulate a multiplicative program *linearly exactly* as follows, referring to it as the Nested reformulation:

$$\max_{\boldsymbol{x},\boldsymbol{y},\hat{\boldsymbol{z}},\hat{\boldsymbol{y}},\boldsymbol{c},\boldsymbol{v},\boldsymbol{u}} \text{ or } \min \quad \sum_{j\in J_p^z} 2^j \hat{z}_{p,j}$$

$$\text{s.t.} \quad (3),\ (4),\ (5),$$

$$y_i = \sum_{j\in J_i^y} 2^j \hat{y}_{i,j}, \qquad i\in I\setminus\{1\}, \tag{12}$$

$$y_1 = \sum_{j\in J_1^z} 2^j \hat{z}_{1,j}, \tag{13}$$

$$u_{i,j,j'} \geq \hat{y}_{i,j} + \hat{z}_{i-1,j'} - 1, \qquad i\in I\setminus\{1\},\ j\in J_i^y,\ j'\in J_{i-1}^z, \tag{14}$$

$$u_{i,j,j'} \leq \hat{y}_{i,j}, \qquad i\in I\setminus\{1\},\ j\in J_i^y,\ j'\in J_{i-1}^z, \tag{15}$$

$$u_{i,j,j'} \leq \hat{z}_{i-1,j'}, \qquad i\in I\setminus\{1\},\ j\in J_i^y,\ j'\in J_{i-1}^z, \tag{16}$$

$$v_{i,j} = \sum_{\substack{j'\in J_i^y,\, j''\in J_{i-1}^z:\\ j'+j''=j}} u_{i,j',j''}, \qquad i\in I\setminus\{1\},\ j\in J_i^z, \tag{17}$$

$$c_{i,0} = 0, \qquad i\in I\setminus\{1\}, \tag{18}$$

$$2c_{i,j} \geq v_{i,j-1} + c_{i,j-1} - 1, \qquad i\in I\setminus\{1\},\ j\in J_i^z\setminus\{0\}, \tag{19}$$

$$2c_{i,j} \leq v_{i,j-1} + c_{i,j-1}, \qquad i\in I\setminus\{1\},\ j\in J_i^z\setminus\{0\}, \tag{20}$$

$$\hat{z}_{i,j} = v_{i,j} + c_{i,j} - \begin{cases} 2c_{i,j+1}, & j < n_i^z - 1, \\ 0, & j = n_i^z - 1, \end{cases} \qquad i\in I\setminus\{1\},\ j\in J_i^z, \tag{21}$$

$$u_{i,j,j'} \in [0,1], \qquad i \in I \setminus \{1\},\ j \in J_i^y,\ j' \in J_{i-1}^z, \tag{22}$$

$$v_{i,j}, c_{i,j} \in \mathbb{N}_0, \qquad i \in I \setminus \{1\},\ j \in J_i^z, \tag{23}$$

$$\hat{y}_{i,j} \in \{0,1\}, \qquad i \in I \setminus \{1\},\ j \in J_i^y, \tag{24}$$

$$\hat{z}_{i,j} \in \{0,1\}, \qquad i \in I,\ j \in J_i^z. \tag{25}$$

Note that Constraints (14)–(17) are equivalent to computing the *cyclic convolution* (i.e., Equation (7)) using a McCormick relaxation. The u variables will naturally take binary values. Constraints (18)–(20) are added to capture the carries (i.e., they are the linear form of Equation (8)), and Constraints (21) compute the binary-encoded product. Observe that the way we interleave the product (2) into

$$\underbrace{\underbrace{(\ldots \underbrace{(\underbrace{(y_1)}_{=z_1}\, y_2)}_{=z_2} \ldots y_{p-1})}_{=z_{p-1}}\, y_p)}_{=z_p}$$

raises an interesting side question: what is the *optimal* way of reindexing the y variables? We leave this question as a future research direction and we do not apply reindexing in this study. However, the following proposition shows that reindexing may result in Nested reformulations with different numbers of variables (and constraints) if $n_1^y = \ldots = n_p^y$ does not hold. The proof of this and all the following propositions in this study can be found in Appendix A.

PROPOSITION 1. For a given reindex $\sigma : (1,\ldots,p) \to (1,\ldots,p)$, the number of u variables, i.e., continuous variables, in the Nested reformulation is

$$\sum_{i=2}^{p} \left(n_{\sigma(i)}^y \left(\left\lfloor \sum_{j=1}^{i-1} \log_2 \left(\bar{y}_{\sigma(j)} \right) \right\rfloor + 1 \right) \right),$$

and the total number of v and c variables, i.e., integer variables, in the Nested reformulation is

$$2 \left(\sum_{i=2}^{p} \left(\left\lfloor \sum_{j=1}^{i} \log_2 \left(\bar{y}_{\sigma(j)} \right) \right\rfloor + 1 \right) \right).$$

3.1.2. Products of length p: the *Altogether* reformulation Interleaving the product (2) into products of length two, while convenient, is not necessary. We may also reformulate the product *altogether*. A motivating example is shown in Table 2 where we carry out the *generalized* FFT-based multiplication of the three integers $y_1 = y_2 = y_3 = 7$.

Table 2 Generalized FFT-based multiplication of the three integers $y_1 = y_2 = y_3 = 7$

j	8	7	6	5	4	3	2	1	0
$\hat{y}_{1,j}$	0	0	0	0	0	0	1	1	1
$\hat{y}_{2,j}$	0	0	0	0	0	0	1	1	1
$\hat{y}_{3,j}$	0	0	0	0	0	0	1	1	1
							1	1	1
						1	1	1	
					1	1	1		
						1	1	1	
					1	1	1		
				1	1	1			
					1	1	1		
				1	1	1			
			1	1	1				
v_j	0	0	1	3	6	7	6	3	1
c_j	1	2	4	5	5	3	1	0	0
$\hat{z}_j$	1	0	1	0	1	0	1	1	1
$y_1 y_2 y_3 = 2^8 + 2^6 + 2^4 + 2^2 + 2^1 + 2^0$									

We are now going to implement the *generalized* FFT-based multiplication as constraints to reformulate a multiplicative program. Let

- $\overline{z} = \prod\limits_{i \in I} \overline{y}_i$ denote the upper bound on $z = \prod\limits_{i \in I} y_i$,
- $n^z = \lfloor \log_2(\overline{z}) \rfloor + 1$ denote the number of binary variables needed to encode z,
- $J^z = \{0, \ldots, n^z - 1\}$,
- $\mathcal{P}_k$, $k \in J^z$, denote the set of two-dimensional sets whose j-indices add up to k:

$$\mathcal{P}_k = \left\{ \{(1, j_1),\, (2, j_2),\, \ldots,\, (p, j_p)\} :\ j_i \in J_i^y,\ i \in I;\ \sum_{i \in I} j_i = k \right\},$$

As an example, consider Table 2 and let $p = 3$ and $\bar{y}_i = y_i = 7$ for $i \in I$. We then have $n_i^y = 3$ and $J_i^y = \{0, 1, 2\}$ for $i \in I$. For $\bar{z} = \prod_{i \in I} \bar{y}_i = 343$, we then have $n^z = 9$ and $J^z = \{0, \ldots, 8\}$. Therefore,

$$\mathcal{P}_0 = \{\{(1,0),(2,0),(3,0)\}\},$$

$$\mathcal{P}_1 = \{\{(1,1),(2,0),(3,0)\},\ \{(1,0),(2,1),(3,0)\},\ \{(1,0),(2,0),(3,1)\}\},$$

$$\begin{aligned}\mathcal{P}_2 = \{&\{(1,2),(2,0),(3,0)\},\ \{(1,0),(2,2),(3,0)\},\ \{(1,0),(2,0),(3,2)\},\ \{(1,1),(2,1),(3,0)\},\\ &\{(1,1),(2,0),(3,1)\},\ \{(1,0),(2,1),(3,1)\}\},\\ &\vdots\\ \mathcal{P}_8 = \emptyset&.\end{aligned}$$

Note that, intuitively speaking, each $k \in J^z$ basically represents the index of a column of values (which are all 1's in this example) in the large middle section of Table 2. Specifically, index $k = 0$ is the index of the column on the rightmost side and $k = 8$ is the index of the column on the leftmost side. Since the number of values reported in the middle section for index $k = 0$ is one, $\mathcal{P}_0$ will only have one element. Similarly, since the number of values reported in the middle section for index $k = 1$ is three, $\mathcal{P}_1$ will only have three elements.

- $\mathcal{P} = \bigcup_{k \in J^z} \mathcal{P}_k$,
- $\hat{z}_j \in \{0,1\}$, $j \in J^z$, denote the binary variables to encode z,
- $u_P \in [0,1]$, $P \in \mathcal{P}$, be a continuous variable that is naturally binary and that encodes the $\hat{y}_{i,j}$ products, $(i,j) \in P$,
- $v_j \in \mathbb{N}_0$, $j \in J^z$, be an integer variable that represents the j-th column sum of the u variables,
- $c_j \in \mathbb{N}_0$, $j \in J^z$, be an integer variable that represents the carried value from column $j-1$ to column j.

We now reformulate a multiplicative program *linearly exactly* as follows, referring to it as the Altogether reformulation:

$$\max_{\boldsymbol{x},\boldsymbol{y},\hat{\boldsymbol{z}},\hat{\boldsymbol{y}},\boldsymbol{c},\boldsymbol{v},\boldsymbol{u}} \text{ or } \min \quad \sum_{j \in J^z} 2^j \hat{z}_j$$

$$\text{s.t.} \quad (3), (4), (5),$$

$$y_i = \sum_{j \in J^y_i} 2^j \hat{y}_{i,j}, \qquad i \in I, \tag{26}$$

$$u_P \geq \sum_{(i,j) \in P} \hat{y}_{i,j} - (p-1), \qquad P \in \mathcal{P}, \tag{27}$$

$$u_P \le \hat{y}_{i,j}, \qquad P \in \mathcal{P},\ (i,j) \in P, \tag{28}$$

$$v_j = \sum_{P \in \mathcal{P}_j} u_P, \qquad j \in J^z, \tag{29}$$

$$c_0 = 0, \tag{30}$$

$$2c_j \ge v_{j-1} + c_{j-1} - 1, \qquad j \in J^z \setminus \{0\}, \tag{31}$$

$$2c_j \le v_{j-1} + c_{j-1}, \qquad j \in J^z \setminus \{0\}, \tag{32}$$

$$\hat{z}_j = v_j + c_j - \begin{cases} 2c_{j+1}, & j < n^z - 1, \\ 0, & j = n^z - 1, \end{cases} \qquad j \in J^z, \tag{33}$$

$$u_P \in [0,1], \qquad P \in \mathcal{P}, \tag{34}$$

$$\hat{z}_j \in \{0,1\},\ \text{and}\ v_j, c_j \in \mathbb{N}_0, \qquad j \in J^z, \tag{35}$$

$$\hat{y}_{i,j} \in \{0,1\}, \qquad i \in I,\ j \in J_i^y. \tag{36}$$

Proposition 2. The number of u variables, i.e., continuous variables, in the Altogether reformulation is

$$\prod_{i \in I} n_i^y,$$

and the total number of v and c variables, i.e., integer variables, in the Altogether reformulation is

$$2\left(\left\lfloor \sum_{i \in I} \log_2\left(\bar{y}_i\right) \right\rfloor + 1\right) = 2n^z.$$

Comparing Propositions 1 and 2 shows that the Nested and Altogether reformulations have very different properties in terms of the number of variables (and constraints). Let $L := \max\{n_1^y, \ldots, n_p^y\}$ and assume that the size of a multiplicative program is polynomially bounded. In that case, the size of its Nested reformulation is polynomially bounded, too, as the number of its u variables is $O(L^2p^2)$ and the total number of its c and v variables is $O(2Lp^2)$. However, the size of the Altogether reformulation is exponential as the number of its u variables is $O(L^p)$ and the total number of its c and v variables is $O(2Lp)$.

3.2. Search mechanisms

To optimize the Nested and Altogether reformulations, we propose two search mechanisms, which in this section we refer to as *One-shot* and *Bitwise*.

3.2.1. One-shot search mechanism In this search mechanism, we directly solve the Nested or Altogether reformulation, which are (mixed) integer linear programs, using an off-the-shelf solver such as CPLEX or Gurobi. The advantage of this search mechanism is that it only solves one optimization problem to compute the optimal values of all bits of the multiplicative objective function. Its disadvantage is that it does not resolve the curse of multiplication.

3.2.2. Bitwise search mechanism Without loss of generality, in this section, we explain the Bitwise search mechanism only in the context of the Altogether reformulation. To apply it in the context of the Nested reformulation, we should add the subscript p to some notations used in this section. Specifically, any instance of n^z, J^z, $\hat{z}_i$, $\hat{z}^*_i$, and $\hat{z}^t_i$ should be changed to n^z_p, J^z_p, $\hat{z}_{p,i}$, $\hat{z}^*_{p,i}$, and $\hat{z}^t_{p,i}$, respectively.

In the Bitwise search mechanism, we compute the optimal values of bits one by one from the most significant one to the least significant one. In each iteration, the optimal value of one bit will be computed and will be fixed for future iterations. The advantage of this search mechanism is that it fully resolves the curse of multiplication. Its disadvantage is that one (mixed) integer integer program with the optimal objective value of either zero or one must be solved for each bit. Specifically, for the Altogether reformulation, the following objective function must be optimized at iteration $t \in J^z$,

$$\max \text{ or } \min \quad \hat{z}_{n^z-1-t},$$

and the following constraints should be added,

$$\hat{z}_{n^z-1-t'} = \hat{z}^*_{n^z-1-t'} \qquad\qquad t' \in J^z : \ t' < t,$$

where $\hat{z}^*_{n^z-1-t'}$ is the optimal value of bit $\hat{z}_{n^z-1-t'}$ obtained in each iteration t'. We now make a few observations.

Observation 1. In the Bitwise search mechanism, in each iteration, one feasible solution will be naturally found for the multiplicative program. Therefore, a global primal bound can be computed for a multiplicative program after each iteration.

To understand Observation 1, consider the Altogether reformulation and let $\hat{\boldsymbol{z}}^t := (z_0^t, \dots, z_{n^z-1}^t)$ be the value of all bits obtained in iteration $t \in J^z$. Since at iteration $t \in J^z$, the optimal value of bit $\hat{z}_{n^z-1-t}$ will be obtained, we have $\hat{z}^*_{n^z-1-t} = \hat{z}^t_{n^z-1-t}$. Moreover, since $\hat{\boldsymbol{z}}^t$ is feasible $\sum_{i \in J^z} 2^i \hat{z}_i^t$ is a global primal bound for the optimal objective value of the multiplicative program.

Observation 2. In the Bitwise search mechanism, in each iteration, a global dual bound can be obtained by setting the value of not-yet-explored bits to their *ideal* values.

To understand Observation 2, note that for mMPs, the ideal value, denoted by v^*, for any arbitrary bit is zero, i.e., $v^* = 0$, as the problem is in the form of minimization (but that may not be feasible). Similarly, for MMPs, the ideal value for any arbitrary bit is one, i.e., $v^* = 1$, as the problem is in the form of maximization (but that, too, may not be feasible). Hence, for the Altogether reformulation, after obtaining $\hat{\boldsymbol{z}}^t$ in iteration $t \in J^z$, we know that

$$\sum_{t'=0}^{t} 2^{n^z-1-t'} \hat{z}^t_{n^z-1-t'} + \sum_{t'=t+1}^{n^z-1} 2^{n^z-1-t'} v^*$$

is a global dual bound for the optimal objective value of the multiplicative program.

Observation 3. In the Altogether reformulation, after obtaining $\hat{\boldsymbol{z}}^t$ in iteration $t \in J^z$, if $\hat{z}^t_{n^z-1-(t+1)} = v^*$, then iteration $t+1$ can be skipped as $\hat{\boldsymbol{z}}^t$ is optimal for iteration $t+1$, too, i.e., we can set $\hat{\boldsymbol{z}}^{t+1} = \hat{\boldsymbol{z}}^t$.

Due to the importance of Observation 3, when implementing the Bitwise search mechanism, we assume that the observation is employed. In other words, we assume that in the basic version of the Bitwise search mechanism, Observation 3 is included for removing redundant calculations and reducing the number of optimization problems needed to be solved. In addition to the basic version of Bitwise, in this study, we consider two additional variations of the Bitwise search mechanism which we will present next.

Bitwise + Full index set Cut (Bitwise + F-cut). This is the basic version of the Bitwise search mechanism with one additional cut (referred to as F-cut) that needs to be added if an optimization problem is solved at iteration t. The *full index set* at iteration t is the set of all indices of not-yet-explored bits whose values in the solution found in iteration $t-1$ are not equal to v^*, i.e., the ideal value. Note that because of Observations 1 and 3, the solution found at iteration $t-1$ corresponds to the best global primal bound known at the beginning of iteration t. For the Altogether reformulation, the full index set at iteration t, denoted by $\mathcal{F}^t$, can be defined as

$$\mathcal{F}^t = \big\{ i \in \{0,\dots,n^z-1-t\} : \ \hat{z}_i^{t-1} \neq v^* \big\}.$$

Observation 4. If the solution found at iteration $t-1$ is not globally optimal for the multiplicative program, then at least one of the bits with indices in $\mathcal{F}^t$ should take its ideal value.

Based on Observation 4, for the Altogether reformulation, the F-cut can be defined as

$$v^* \sum_{i \in F^t} \hat{z}_i + (1-v^*) \sum_{i \in F^t} (1-\hat{z}_i) \geq 1.$$

Observation 5. If after adding the F-cut at iteration t, the optimization problem that needs to be solved at iteration t becomes infeasible, then the solution found at iteration $t-1$ is globally optimal and the Bitwise search can terminate immediately.

Bitwise + Partial index set cut (Bitwise + P-cut). This variation is similar to Bitwise + F-cut. The main difference is that F-cut will be replaced by a different cut, referred to as P-cut. At the beginning of iteration t, let α^t be the index of the most significant not-yet-explored bit whose value in the solution found in iteration $t-1$ is equal to v^*, i.e., the ideal value. In other words, for the Altogether reformulation α^t can be defined as

$$\alpha^t = \max \big\{ i \in \{0,\dots,n^z-1-t\} : \ \hat{z}_i^{t-1} = v^* \big\}.$$

The *partial index set*, denoted by $\widetilde{\mathcal{F}}^t$, is the set of all elements in $\mathcal{F}^t$ that are greater than α^t, i.e.,

$$\widetilde{\mathcal{F}}^t = \big\{ i \in \mathcal{F}^t : i > \alpha^t \big\}.$$

Using $\widetilde{\mathcal{F}}^t$, for the Altogether reformulation, the P-cut can be defined as

$$v^* \sum_{i \in \widetilde{\mathcal{F}}^t} \hat{z}_i + (1-v^*) \sum_{i \in \widetilde{\mathcal{F}}^t} (1-\hat{z}_i) \geq 1.$$

Note that $\widetilde{\mathcal{F}}^t$ is a subset of sequential most significant not-yet-explored bits whose values are not ideal in the solution obtained in iteration $t-1$. Hence, the P-cut is designed to ensure that at least one of these bits will take its ideal value in iteration t if possible. However, if it is not possible, then we have the following observation that helps us skip some iterations.

Observation 6. If after adding the P-cut at iteration t, the optimization problem that needs to be solved at iteration t becomes infeasible, then the optimal value of the bits with indices in $\widetilde{\mathcal{F}}^t$ cannot be ideal. In other words, their values will surely be $1-v^*$. This implies that for the Altogether reformulation, we can set $\hat{\boldsymbol{z}}^{t'} = \hat{\boldsymbol{z}}^{t-1}$ for all $t' \in \{t, t+1, \ldots, t+|\widetilde{\mathcal{F}}^t|-1\}$, and all iterations $\{t, t+1, \ldots, t+|\widetilde{\mathcal{F}}^t|-1\}$ can be skipped.

3.3. Warm-start enhancements for minimum multiplicative problems

Similar to the previous section, again without loss of generality, we explain our enhancements only in the context of the Altogether reformulation. However, the same enhancements can be applied to the Nested reformulation by simply adding the subscript p to some notations used in this section. Specifically, any instance of n^z and $\hat{z}^w_i$ should be replaced by n^z_p and $\hat{z}^w_{p,i}$, respectively. With this in mind, suppose that before starting a search mechanism (One-shot or Bitwise), a feasible solution is known for a multiplicative program that can be used for warm-starting. We denote the objective bit values of this feasible solution by $\hat{\boldsymbol{z}}^w := (\hat{z}^w_0, \ldots, \hat{z}^w_{n^z-1})$ for the Altogether reformulation. Moreover, we denote the most significant bit in $\hat{\boldsymbol{z}}^w$ that has not taken its ideal value by β^w, i.e.,

$$\beta^w := \max\left\{ i \in \{0, \ldots, n^z-1\} : \; \hat{z}^w_{n^z-1} \neq v^* \right\}$$

for the Altogether reformulation.

Observation 7. In the Altogether reformulation, the optimal values of bits with indices $\{\beta^w + 1, \beta^w + 2, \ldots, n^z - 1\}$ are equal to the ideal value, i.e., $\hat{z}_i = v^*$ for all $i \in \{\beta^w + 1, \beta^w + 2, \ldots, n^z - 1\}$.

Observation 7 is important as it suggests that the optimal value of some of the most significant bits can be determined in advance if a feasible solution is known in advance. In practice, this observation is more likely to be effective for mMPs. This is because the maximum number of bits required to represent the optimal objective value of a multiplicative program, i.e., n^z (for the Altogether formulation), is determined based on the product of the upper bounds on $y_1,\dots,\ y_p$. Therefore, since mMPs are in the form of minimization, many of the most significant bits naturally take the optimal value of zero. We hence suggest to employ Observation 7 only for mMPs. Next, we present two simple ways of computing a good initial feasible solution for mMPs.

3.3.1. Min-Min appraoch Our first proposed approach minimizes the minimum of $y_1,\dots,y_p$ over the feasible set. The following (mixed) integer linear program needs to be solved:

$$\min_{\boldsymbol{x},\boldsymbol{y},\boldsymbol{\lambda},\theta} \big\{\theta : (3),\ (4),\ (5),\ \theta \geq y_i - M\lambda_i \quad i\in I,\ \sum_{i\in I}\lambda_i = p-1,\ \lambda_i\in\{0,1\} \quad i\in I\big\},$$

where M is a sufficiently large number. We denote the vector of optimal values for the y-variables by $\boldsymbol{y}^w$. The binary-encoded version of $\prod_{i\in I} y_i^w$ generates $\hat{\boldsymbol{z}}^w$.

3.3.2. Indirect Min-Min approach In our second approach, we first minimize each y_i for $i\in I$ separately and then choose the solution of the problem that results in the smallest objective value for mMP. In other words, for each $i\in I$, we solve

$$\arg\min_{\boldsymbol{x},\boldsymbol{y}} \big\{y_i : (3),\ (4),\ (5)\big\},$$

and denote the vector of optimal values for the y-variables by $\boldsymbol{y}^{w,i}$. The binary-encoded version of

$$\min\left\{\prod_{i\in I} y_i^{w,1},\dots,\prod_{i\in I} y_i^{w,p}\right\}$$

generates $\hat{\boldsymbol{z}}^w$.

3.4. Continuous cases

In all previous cases, we assumed that all y-variables are integer. In this section, we explain how our approach can be used for approximating multiplicative programs with $p_C \neq 0$ (with any desirable level of precision). Suppose that there exists an $i \in I$ such that y_i can only take fractional values. Suppose further that users are interested in only $\beta \in \mathbb{N}_0$ digits after the decimal point of the value of y_i. The following two steps should be applied in order to modify the Nested or the Altogether reformulation:

- Step 1: We first find the equation corresponding to y_i from Constraint (3). We then multiply its right-hand side by 10^β. Note that to avoid running into numerical issues and removing any solutions, we keep the declaration of y_i as continuous variables, i.e., we do not change it to integer. As an aside, we note that Constraint (3) appears in both the Nested and the Altogether reformulation. Hence, both reformulations will be modified in this step.
- Step 2: We change the equation corresponding to y_i in Constraints (12), (13), and (26) from equality to inequality. Specifically, we change '$=$' to '$\geq$' for the maximization instances. Moreover, we change '$=$' to '$\leq$' for the minimization instances. These inequalities ensure that the optimal binary-encoded value of y_i represents $\lfloor y_i \rfloor$ and $\lceil y_i \rceil$ in MMPs and mMPs, respectively. In other words, our approximation technique is a lower approximation for MMPs and an upper approximation for mMPs.

4. Extreme examples

As mentioned earlier, the Bitwise search mechanism is the only strategy (in this paper) that can truly resolve the curse of multiplication. Moreover, the Nested reformulation keeps the size of the problem polynomially bounded while the Altogether reformulation increases the size of the problem exponentially (as p increases). To solve instances with large values of p, the only suitable approach is to employ the Nested reformulation when being combined with the Bitwise search mechanism. In order to testify this claim, in this section, we compare the performance of our approach, i.e., Nested reformulation + Bitwise search mechanism, with an off-the-shelf solver on two instances where the curse of multiplication is likely to arise.

Both instances have 20 binary variables, i.e., $n_C = 0$ and $n_I = 20$, and 10 constraints in the space of x-variables, i.e., $m = 10$. However, one instance has $p = 15$ and the other has $p = 20$. Interested readers may refer to Appendix B to find the fully defined instances. As an aside, we note that although these instances look small, simply generating the Altogether reformulation can consume a significant amount of time and memory space due to $p \in \{15, 20\}$. In fact we could not load the Altogether reformulations of either one of these two instances in a computing platform with 128GB RAM. However, the Nested reformulations require a negligible amount of time and memory space and that is why we have used it in this section.

Table 3 Results for the instance with $p = 15$

	Min		Max	
	Objective value	Time (s)	Objective value	Time (s)
Bitwise	37,881,049,842,155,520	669.21	13,426,599,939,480,000,000	2107.71
GRB	131,262,150,868,992,000	0.07	2,456,521,675,442,388,480	0.05

We solve each instance both in the form of minimization and maximization using our approach, i.e., Nested reformulation + Bitwise search mechanism. Additionally, we solve each instance using a powerful commercial off-the-shelf solver, Gurobi version 9.0.2 (which we refer to as GRB), to conduct comparisons. As previously mentioned in Section 2, in order to solve multiplicative programs using GRB, an MMP can be reformulated as a (mixed integer) SOCP and an mMP can be reformulated as a nonconvex optimization program, specifically, a nonconvex quadratically constrained quadratic program. Hence, we employ GRB SOCP for the maximization form and GRB Nonconvex for the minimization form. For consistency, when implementing our algorithm, we also set GRB as the default solver for solving mixed integer linear programs arising during the course of our algorithm. We impose a time limit of 259,200 seconds (3 days) and set the optimality gap tolerance to zero for all solution approaches. The results are shown in Tables 3 and 4 for $p = 15$ and $p = 20$, respectively.

Table 4 Results for the instance with $p = 20$

	Min		Max	
	Objective value	Time (s)	Objective value	Time (s)
Bitwise	322,275,494,202,543,737,864,192	259,200.00	65,290,849,092,115,078,053,888,000	259,200.00
GRB	*Infeasible*	0.00	–	259,200.00

In Table 3, we observe that GRB claims optimality within a fraction of a second. However, obviously, the objective values reported are far from the optimal values reported by the Bitwise approach. This is an indication that GRB has faced numerical issues. In Table 4, we observe that GRB Nonconvex (for minimization) immediately reports infeasibility and GRB SOCP (for maximization) does not report any feasible solution within the time limit. However, our Bitwise approach found feasible solutions for both cases reporting $4 \times 10^{-7}\%$ and 98.6% optimality gap for minimization and maximization, respectively. We note that the 98.6% optimality gap reported for the maximization case is due to the magnitude of the objective value.

5. Computational study

In this computational study, we compare the performance of our proposed algorithms with GRB SOCP and GRB Nonconvex when solving MMPs and mMPs for small values of p, respectively. All solution methods as well as our instance generator are implemented in C++ (the source files are available at https://github.com/paymanghasemi/Multiplicative-Programs-by-Binary-encoding-the-Multiplication-Operation and the '.lp' format of the instances are available at https://usf.box.com/s/1u6xesylwufybxj8fjwyefslcxw77nyt). For consistency, when implementing our algorithms, we also set GRB as the default solver for solving mixed integer linear programs arising during the course of our algorithms. All computational experiments are conducted on a Dell PowerEdge R360 with two Intel Xeon E5-2650 2.2 GHz 12-core processors (30MB), 128GB RAM, the RedHat Enterprise Linux 7.0 operating system, using a single thread. We set the optimality gap tolerance of GRB to zero in all experiments. Moreover, a time limit of 3,600 seconds (1 hour) is imposed for solving each instance using each solution method.

We generate 900 instances for this computational study through the procedure described in Appendix C. Specifically, we generate 300 binary instances that can be used in both maximization and minimization forms, 300 continuous maximization instances, and 300 continuous minimization instances. Our computational study is based on 5,400 runs where each run means solving an instance by a specific solution method. These 5,400 runs are divided (not equally) into four experiments:

- **Experiment 1.** The first experiment focuses on identifying the best search mechanism. We conduct this experiment on only instances with $p = 2$ as the Altogether and Nested reformulations are the same for such instances. Since our solution methods are only approximate algorithms for continuous instances, we only focus on pure binary instances for the first experiment. We show that One-shot is the fastest search mechanism. Hence, in the remaining experimental settings, One-shot is set as the default search mechanism for our solution approaches.
- **Experiment 2.** In the second experiment, we focus on binary instances in minimization form with $p \in \{2, 3\}$ to identify whether the proposed warm-start techniques are effective and, if so, which one performs the best. Note that we do not introduce any enhancements for the maximization instances (in this paper) and that is why the focus of this experiment is only on minimization instances. In this experiment, we use the Nested reformulation (combined with the One-shot search mechanism). Overall, based on the results of the second experiment, we show that 'Indirect Min-Min' performs best. Therefore, in the remaining experiments, 'Indirect Min-Min' is the default enhancement technique when solving minimization instances.
- **Experiment 3.** In the third experiment, we compare the performance of the Nested and Altogether reformulations (when being combined with the One-shot search mechanism and the Indirect Min-Min enhancement) with the performance of GRB on all pure binary instances with $p \in \{2, 3, 4\}$. Specifically, for pure binary maximization instances, GRB SOCP is the benchmark for comparison, and for pure binary minimization instances, GRB Nonconvex is the benchmark for comparison. We show that GRB SOCP performs best for maximization instances while for minimization instances, the Nested reformulation is the dominant approach.
- **Experiment 4.** In the fourth experiment, we repeat Experiment 3 but for all continuous instances with $p \in \{2, 3, 4\}$. The only difference is that due to the poor performance of the Altogether reformulation, we do not use it in this experiment. Moreover, as our proposed approaches are approximations for continuous instances, we consider three multipliers $\{10^0, 10^2, 10^4\}$ for transforming the continuous y-variables into integers, i.e., $\beta \in \{0, 2, 4\}$. We show that GRB

SOCP performs best for maximization instances while for minimization instances, the Nested reformulation is the dominant approach.

In light of the above, the following are the solution approaches (in addition to GRB SOCP and GRB Nonconvex) that we use in our experiments for binary instances:

- N-O: Nested, One-shot,
- N-B: Nested, Bitwise,
- N-B+F: Nested, Bitwise + F-cut,
- N-B+P: Nested, Bitwise + P-cut,
- N-O-Imm: Nested, One-shot, Indirect Min-Min,
- N-O-mm: Nested, One-shot, Min-Min,
- A-O: Altogether, One-shot,
- A-O-Imm: Altogether, One-shot, Indirect Min-Min.

Specifically, N-O, N-B, N-B+F, and N-B+P are used in Experiment 1. Note that the Nested and Altogether reformulations are indeed the same when dealing with instances of $p = 2$. Hence, the term 'Nested' can be replaced by 'Altogether' for instances with $p = 2$. In Experiment 2, N-O, N-O-Imm, and N-O-mm are used. In Experiment 3, N-O, A-O, and GRB SOCP are used for maximization instances while N-O-Imm, A-O-Imm, and GRB Nonconvex are used for minimization instances. The following are the solution approaches (in addition to GRB SOCP and GRB Nonconvex) that we used in our experiments for continuous instances:

- N-O-M: N-O with M as multiplier from the set $\{10^0, 10^2, 10^4\}$,
- N-O-Imm-M: N-O-Imm with M as multiplier from the set $\{10^0, 10^2, 10^4\}$.

Specifically, in Experiment 4, N-O-10^0, N-O-10^2, N-O-10^4, and GRB SOCP are used for maximization instances while N-O-Imm-10^0, N-O-Imm-10^2, N-O-Imm-10^4, and GRB Nonconvex are used for minimization instances.

Table 5 shows a high-level summary of the computational results of all 5,400 runs. It is worth mentioning that the result tree shown in Figure 1 (given in the introduction) is generated based on

Table 5 High-level summary of computational results

Direction			*p*					
			2		3		4	
Maximize Variables	Binary		# Runs:	600	# Runs:	300	# Runs:	300
			# Instances:	100	# Instances:	100	# Instances:	100
			GRB SOCP	85.00%	GRB SOCP	82.00%	GRB SOCP	87.00%
			Time outs	14.00%	Time outs	18.00%	Time outs	13.00%
			N-O	1.00%				
	Continuous		# Runs:	400	# Runs:	400	# Runs:	400
			# Instances:	100	# Instances:	100	# Instances:	100
			GRB SOCP	95.00%	GRB SOCP	85.00%	GRB SOCP	92.00%
			N-O-10^0	4.00%	N-O-10^0	8.00%	Time outs	7.00%
			N-O-10^4	1.00%	Time outs	6.00%	N-O-10^4	1.00%
					N-O-10^4	1.00%		
Minimize Variables	Binary		# Runs:	900	# Runs:	600	# Runs:	300
			# Instances:	100	# Instances:	100	# Instances:	100
			N-O-Imm	76.17%	N-O-Imm	75.00%	N-O-Imm	40.00%
			N-O	17.17%	Time outs	12.00%	Time outs	36.00%
			GRB Nonconvex	3.33%	A-O-Imm	11.00%	GRB Nonconvex	24.00%
			N-O-mm	3.33%	GRB Nonconvex	1.00%		
					N-O	1.00%		
	Continuous		# Runs:	400	# Runs:	400	# Runs:	400
			# Instances:	100	# Instances:	100	# Instances:	100
			N-O-Imm-10^0	94.00%	N-O-Imm-10^0	100.00%	N-O-Imm-10^0	89.00%
			N-O-Imm-10^2	3.00%			GRB Nonconvex	7.00%
			GRB Nonconvex	2.00%			N-O-Imm-10^2	3.00%
			N-O-Imm-10^4	1.00%			Time outs	1.00%

Table 5. In this table, we classify the problem variation by the *direction* (maximize, minimize), the *variables* (binary, continuous), and $p \in \{2,3,4\}$. Hence, in total there are $2 \times 2 \times 3 = 12$ problem variations. For each problem variation, we provide the number of unique runs and instances in Table 5. Below that, we list the solution methods and their share of instances for which they had the shortest solution time. For some instances, there may be a tie between two or more solution methods, which explains the fractional percentages. For example, for the maximization, binary, and $p = 2$ case, GRB SOCP had the shortest solution time for 85% of instances, followed by 14% time outs and 1% One-shot. Solution methods that never had the shortest solution time for any instance (i.e., 0%) are omitted. It is immediately clear that GRB SOCP performs best for maximization problems. For minimization problems, however, the solution methods introduced in this paper outperform the off-the-shelf benchmarks. Specifically, for binary problems, N-O-Imm and N-O perform best for $p = 2$. For $p = 3$, the top performers are N-O-Imm and A-O-Imm. It is only for $p = 4$ that the instances either time out or can only be solved by GRB Nonconvex or N-O-Imm. For continuous problems, N-O-Imm-10^0 approximations dominate the other approaches.

To see more details of Experiments 1–4, interested readers may refer to Appendix D. Moreover, there is a 185-page supplementary PDF document (along with its corresponding

CSV file) available at https://github.com/paymanghasemi/Multiplicative-Programs-by-Binary-encoding-the-Multiplication-Operation for details on every one of the 5,400 runs in this study.

6. Final remarks

In this paper, we addressed the question of how the curse of multiplication can be resolved when solving multiplicative programs. Specifically, we showed how the multiplication operation itself can be binary-encoded as an integer linear program following the procedure that a computer uses internally. We developed two types of binary-encoded reformulations for a multiplicative program: the Nested reformulation and the Altogether reformulation. Assuming that a multiplicative program has a compact (i.e., polynomially bounded) size, we proved that the Nested reformulation will also have a compact size. However, that is not the case for the Altogether reformulation. We also introduced two search mechanisms, One-shot and Bitwise, with several enhancement techniques for solving the proposed reformulations. One-shot is suitable for multiplicative programs with small values of p, while Bitwise is suitable for multiplicative programs with large values of p. Therefore, the combination of the Nested reformulation and the Bitwise search mechanism is the first approach (to the best of our knowledge) that can fully resolve the curse of multiplication in multiplicative programs. Although our focus has been mainly on resolving the curse of multiplication, surprisingly, we observed that the Nested reformulation combined with the One-shot search mechanism completely outperforms standard solvers on minimization instances, regardless of the value of p. However, for maximization instances with small values of p, standard solvers perform best. This difference can be explained by the fact that maximization instances are different from minimization instances in nature, i.e., continuous mMPs are NP-hard while continuous MMPs are polynomially solvable.

We hope that the simplicity of our proposed solution approaches and their promising results encourage more researchers to study multiplicative programs. Specifically, many different reformulations and/or search mechanisms can be developed by combining ideas behind those presented in this paper. Therefore, studying such reformulations and search mechanisms both theoretically and computationally could be valuable.

Acknowledgments

We thank Dr. Pablo Garcia-Herreros (Aurubis AG, Hamburg, Germany) for the fruitful discussions in 2018 that led to the Nested and Altogether formulations as well as some promising preliminary computational results.

References

Abouei Ardakan M, Sima M, Zeinal Hamadani A, Coit DW (2016) A novel strategy for redundant components in reliability–redundancy allocation problems. *IIE Transactions* 48(11):1043–1057.

Ben-Tal A, Nemirovski A (2001) On polyhedral approximations of the second-order cone. *Mathematics of Operations Research* 26(2):193–205.

Benson H, Boger G (2000) Outcome-space cutting-plane algorithm for linear multiplicative programming. *Journal of Optimization Theory and Applications* 104(2):301–322.

Beyer HL, Dujardin Y, Watts ME, Possingham HP (2016) Solving conservation planning problems with integer linear programming. *Ecological Modelling* 328:14–22.

Billionnet A (2013) Mathematical optimization ideas for biodiversity conservation. *European Journal of Operational Research* 231(3):514–534.

Borwein J, Bailey D (2008) *Mathematics by Experiment: Plausible Reasoning in the 21st Century* (A K Peters/CRC Press), 2nd edition.

Caragiannis I, Kurokawa D, Moulin H, Procaccia AD, Shah N, Wang J (2019) The unreasonable fairness of maximum nash welfare. *ACM Transactions on Economics and Computation (TEAC)* 7(3):1–32.

Charkhgard H, Savelsbergh M, Talebian M (2018) A linear programming based algorithm to solve a class of optimization problems with a multi-linear objective function and affine constraints. *Computers & Operations Research* 89:17 – 30, https://doi.org/10.1016/j.cor.2017.07.015.

Dey SS, Gupte A (2015) Analysis of MILP techniques for the pooling problem. *Operations Research* 63(2):412–427.

Dolan ED, Moré JJ (2002) Benchmarking optimization software with performance profiles. *Mathematical Programming* 91(2):201–213.

Failing L, Gregory R (2003) Ten common mistakes in designing biodiversity indicators for forest policy. *Journal of Environmental Management* 68(2):121 – 132.

Feizabadi M, Jahromi AE (2017) A new model for reliability optimization of series-parallel systems with non-homogeneous components. *Reliability Engineering & System Safety* 157:101–112.

Luedtke J, Namazifar M, Linderoth J (2012) Some results on the strength of relaxations of multilinear functions. *Mathematical Programming* 136(2):325–351.

Nash Jr JF (1950) The bargaining problem. *Econometrica: Journal of the Econometric Society* 155–162.

Nicholson E, Possingham HP (2006) Objectives for multiple-species conservation planning. *Conservation Biology* 20(3):871–881.

Saghand PG, Charkhgard H, Kwon C (2019) A branch-and-bound algorithm for a class of mixed integer linear maximum multiplicative programs: A bi-objective optimization approach. *Computers & Operations Research* 101:263–274.

Shao L, Ehrgott M (2014) An objective space cut and bound algorithm for convex multiplicative programmes. *Journal of Global Optimization* 58(4):711–728.

Shao L, Ehrgott M (2016) Primal and dual multi-objective linear programming algorithms for linear multiplicative programmes. *Optimization* 65(2):415–431.

Sierra-Altamiranda A, Charkhgard H, Eaton M, Martin J, Yurek S, Udell BJ (2020) Spatial conservation planning under uncertainty using modern portfolio theory and nash bargaining solution. *Ecological Modelling* 423:109016.

Vazirani VV (2012) The notion of a rational convex program, and an algorithm for the arrow-debreu Nash bargaining game. *Journal of the ACM (JACM)* 59(2):1–36.

Zhang E, Chen Q (2016) Multi-objective reliability redundancy allocation in an interval environment using particle swarm optimization. *Reliability Engineering & System Safety* 145:83–92.

Appendix A: Proofs

Proof of Proposition 1. Observe that in the Nested reformulation, we interleave the product (2) into

$$\underbrace{(\underbrace{(\ldots \underbrace{(\underbrace{(y_{\sigma(1)})}_{=z_1} y_{\sigma(2)})}_{=z_2} \ldots y_{\sigma(p-1)})}_{=z_{p-1}} y_{\sigma(p)})}_{=z_p},$$

where $z_1 = y_{\sigma(1)}$ and $z_i = y_{\sigma(i)} z_{i-1}$ for each $i = 2, \ldots, p$. We note that the u variables are used in Constraints (14)–(16), i.e., the McCormick relaxation constraints for the binary multiplication of $y_{\sigma(i)} z_{i-1}$ for each $i = 2, \ldots, p$. We know that the number of bits required to represent $y_{\sigma(i)}$ is equal to $n^y_{\sigma(i)}$ for each $i = 1, 2, \ldots, p$. We also know that $z_{i-1} \le \prod_{j=1}^{i-1} \bar{y}_{\sigma(j)}$ for each $i = 2, \ldots, p$. Hence, the number of bits required to represent z_{i-1} is $\lfloor \sum_{j=1}^{i-1} \log_2 (\bar{y}_{\sigma(j)}) \rfloor + 1$ for each $i = 2, \ldots, p$. Further, the total number of u variables in the Nested reformulation is

$$\sum_{i=2}^{p} \left(n^y_{\sigma(i)} \left(\left\lfloor \sum_{j=1}^{i-1} \log_2 \left(\bar{y}_{\sigma(j)}\right) \right\rfloor + 1 \right) \right).$$

Similarly, observe from Constraints (17)–(21) that in order to compute z_i, some v variables and c variables should be generated for each $i = 2, \ldots, p$. Specifically, when computing z_i, the number of bits required to represent z_i is the number of v variables (or the number of c variables). Hence, the total number of c and v variables is

$$2 \sum_{i=2}^{p} \left(\left\lfloor \sum_{j=1}^{i} \log_2 \left(\bar{y}_{\sigma(j)}\right) \right\rfloor + 1 \right).$$

Q.E.D.

Proof of Proposition 2. Observe that in the Altogether reformulation, we calculate the product (2) as a whole, i.e.,

$$z = \prod_{i \in I} y_i.$$

We note that the u variables are used in Constraints (27)–(28), i.e., the McCormick relaxation constraints for the binary multiplication of $\prod_{i \in I} y_i$. We know that $z \le \prod_{j \in I} \bar{y}_i$ and that $\bar{y}_i$ requires n^y_i bits to be represented for each $i = 1, \ldots, p$. Therefore, the total number of u variables in the Altogether reformulation is

$$\prod_{i \in I} n^y_i.$$

Similarly, observe from Constraints (29)–(32) that in order to compute z, some v and c variables should be generated. Specifically, when computing z, the number of bits required to represent z is the number of v variables (or the number of c variables). Therefore, the total number of c and v variables is

$$2\left(\left\lfloor \sum_{i\in I} \log_2\left(\bar{y}_i\right)\right\rfloor + 1\right) = 2n^z.$$

Q.E.D.

Appendix B: Extreme examples

In this section, we provide the model of each extreme example in '.lp' format. We do not provide the objective function of the models as it is simply the multiplication of all y-variables. We first provide the model for the extreme example with $p=15$. In this model, $C1$ to $C15$ are equations used to represent y-variables.

$$C1: y_1+8x_1+4x_2+9x_7+4x_8+x_9+x_{10}+6x_{12}-4x_{13}-5x_{17}=18$$

$$C2: y_2-3x_1+6x_6+10x_7+9x_8+10x_9-x_{10}-9x_{11}-3x_{14}+7x_{16}-2x_{19}=20$$

$$C3: y_3+6x_1-7x_2+10x_4-2x_5-10x_6-4x_8-5x_9+9x_{11}-2x_{12}+7x_{14}+7x_{15}-7x_{17}-9x_{19}+x_{20} =34$$

$$C4: y_4+4x_3-10x_5-10x_7+4x_8+3x_9-5x_{12}+10x_{13}+2x_{16}-10x_{17}+8x_{18}-3x_{19}=23$$

$$C5: y_5-8x_1-3x_3-5x_4+3x_5+6x_{10}+x_{12}-4x_{15}+10x_{16}+10x_{18}+6x_{19}=35$$

$$C6: y_6+9x_3-7x_4-10x_6-6x_8+7x_9+8x_{12}-3x_{20}=10$$

$$C7: y_7+x_1-7x_4-4x_{10}+5x_{15}-2x_{17}-3x_{19}=13$$

$$C8: y_8+7x_1-3x_2+4x_3+9x_5-2x_7+7x_8-7x_9-6x_{11}-2x_{13}-10x_{16}+3x_{17}-7x_{18}=16$$

$$C9: y_9-10x_1+5x_2-8x_4-6x_6-4x_9+4x_{10}-6x_{12}+7x_{13}+x_{16}+10x_{17}-10x_{18}+9x_{19}=13$$

$$C10: y_{10}+4x_1-7x_3+4x_6-6x_7+4x_{11}-10x_{12}+2x_{14}+x_{15}-3x_{17}-9x_{19}+10x_{20}=17$$

$$C11: y_{11}-9x_1-7x_2+9x_4-6x_5-10x_7-4x_{11}+2x_{17}+7x_{19}=17$$

$$C12: y_{12}-3x_6+2x_7+x_9+2x_{12}+5x_{16}+7x_{19}=17$$

$$C13: y_{13}-2x_3+9x_4+x_6+8x_7+2x_9+6x_{12}-x_{15}-4x_{16}+x_{18}+9x_{19}-9x_{20}=25$$

$$C14: y_{14}-5x_1+4x_3+4x_5-2x_8-5x_9+4x_{10}-8x_{11}-9x_{13}-9x_{14}-4x_{15}-10x_{16}+7x_{17}-9x_{18} -2x_{20}=1$$

$$C15: y_{15}+x_2-x_8-6x_9-5x_{10}+9x_{11}+7x_{14}+2x_{15}-x_{16}+5x_{20}=24$$

$$C16: 11x_1+11x_5+27x_{10}+4x_{11}+22x_{12}+22x_{15}+24x_{19}+28x_{20}\leq 32$$

$$C17: 27x_2+21x_3+13x_5+7x_7+15x_8+19x_{10}+25x_{11}+3x_{12}+4x_{13}+5x_{16}+15x_{17}\leq 27$$

$$C18: 8x_2+24x_3+15x_7+21x_{11}+16x_{13}+17x_{14}+22x_{16}+15x_{18}+8x_{19}\leq 60$$

$$C19: 26x_1+2x_3+7x_4+16x_5+26x_6+23x_8+17x_9+24x_{11}+30x_{13}+29x_{14}+18x_{17}+15x_{18}+9x_{19} \leq 32$$

$$C20: 21x_2+21x_3+28x_4+11x_5+20x_6+13x_7+5x_9+x_{11}+16x_{12}+11x_{15}\leq 91$$

$$C21: 16x_3+9x_4+4x_5+14x_6+20x_7+6x_9+8x_{15}+x_{16}\leq 80$$

$$C22: 28x_3+17x_6+22x_7+7x_{11}+27x_{14}+29x_{15}+11x_{17}\leq 52$$

$C23: 26x_1 + 10x_3 + 28x_5 + 26x_8 + 27x_{10} + 30x_{12} + 20x_{15} + 5x_{16} + 28x_{17} + 18x_{18} \leq 26$

$C24: 2x_1 + 27x_2 + 30x_3 + 28x_5 + 29x_7 + 22x_9 + 7x_{10} + 29x_{12} + 4x_{13} + 24x_{19} + 4x_{20} \leq 27$

$C25: x_1 + 22x_3 + 9x_5 + 30x_{10} + 12x_{13} + 17x_{14} + 22x_{17} + 5x_{18} + 30x_{19} + 19x_{20} \leq 89$

Binaries

$x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8\ x_9\ x_{10}\ x_{11}\ x_{12}\ x_{13}\ x_{14}\ x_{15}\ x_{16}\ x_{17}\ x_{18}\ x_{19}\ x_{20}$

We now provide the model for the extreme example with $p = 20$. In this model, $C1$ to $C20$ are equations used to represent y-variables.

$C1: y_1 - 10x_1 + 3x_2 + 7x_3 - 8x_4 - 7x_6 - x_7 - 9x_9 - 2x_{10} + 2x_{11} - 10x_{13} - 8x_{14} + 3x_{15} + 8x_{16} + 10x_{17}$
$\quad + 3x_{18} + 10x_{19} - 6x_{20} = 25$

$C2: y_2 + 5x_2 + 6x_5 + 5x_6 - 10x_7 + 6x_{11} + 6x_{12} - 9x_{14} + 6x_{15} + 4x_{17} - 6x_{18} + 8x_{19} - 8x_{20} = 21$

$C3: y_3 + x_3 - 3x_4 + 4x_6 - 7x_8 - 10x_9 + 6x_{10} + 6x_{11} + 4x_{19} + 4x_{20} = 17$

$C4: y_4 + 10x_4 - 3x_7 - 3x_{11} - 8x_{13} + 3x_{14} - 9x_{19} = 16$

$C5: y_5 - 10x_2 + 7x_3 - 8x_5 - 5x_6 - 5x_7 + 2x_8 - 7x_{10} - 4x_{13} + 2x_{17} + 5x_{18} + 7x_{19} + 6x_{20} = 24$

$C6: y_6 - 9x_1 - 8x_3 - 4x_4 - 5x_8 - 8x_{10} + 6x_{13} + 9x_{15} - 3x_{17} - 4x_{19} = 9$

$C7: y_7 - 7x_2 + 5x_8 - 2x_{10} - 7x_{12} - 9x_{15} - 4x_{16} - 8x_{18} + 2x_{19} - 2x_{20} = 14$

$C8: y_8 + 5x_2 + 2x_8 + 5x_{10} - 10x_{11} - 6x_{12} - 2x_{15} + 7x_{16} + 3x_{17} - 6x_{18} - 8x_{19} + 8x_{20} = 19$

$C9: y_9 - 8x_9 + x_{10} - 8x_{15} + 10x_{18} + 8x_{19} + 2x_{20} = 21$

$C10: y_{10} + x_2 - 3x_4 + 4x_5 + 8x_6 + 5x_7 + 6x_8 - 10x_{10} - x_{12} + 7x_{13} - 4x_{15} + 2x_{16} + 8x_{19} - 2x_{20} = 24$

$C11: y_{11} + 9x_1 - 2x_3 + 2x_4 - 8x_5 + 7x_{10} - 3x_{11} - 6x_{14} - 6x_{17} - 8x_{18} = 18$

$C12: y_{12} - 2x_3 + 3x_4 - 8x_5 - x_9 - 8x_{10} + 10x_{13} - x_{14} - 3x_{15} - 5x_{17} - 9x_{19} - 9x_{20} = 20$

$C13: y_{13} + 3x_3 - 6x_4 + 2x_5 + 4x_7 - 5x_8 + 2x_{14} - 4x_{15} - 2x_{16} = 19$

$C14: y_{14} + 8x_5 + 7x_6 - 2x_8 - 6x_{13} + 5x_{17} - 4x_{18} - 2x_{19} - 5x_{20} = 18$

$C15: y_{15} + 6x_4 - 6x_6 + 7x_8 + 5x_{11} - 2x_{17} + 10x_{18} - 9x_{19} + 2x_{20} = 31$

$C16: y_{16} - 6x_1 + 4x_2 - 6x_3 + 9x_4 - 5x_5 + 9x_{12} + 7x_{15} - 8x_{16} + 2x_{17} + 7x_{18} + 10x_{19} + 7x_{20} = 28$

$C17: y_{17} + 6x_1 - 10x_2 - 10x_3 - 5x_6 + 2x_{14} - 10x_{16} - 10x_{19} - 10x_{20} = 9$

$C18: y_{18} - 4x_2 - 5x_5 + x_6 + 7x_7 - 8x_8 + 5x_9 + 6x_{10} - x_{11} - 8x_{12} - 5x_{14} + 3x_{15} + 3x_{18} - 6x_{20} = 16$

$C19: y_{19} + 10x_1 + x_5 - x_6 + 4x_8 + 9x_9 + 10x_{12} + x_{14} - 5x_{17} - 9x_{18} + 4x_{20} = 27$

$C20: y_{20} - 10x_1 + 10x_4 + x_9 + 4x_{10} + 9x_{12} + 4x_{13} + 4x_{16} - 5x_{17} + 10x_{20} = 38$

$C21: 8x_1 + 4x_2 + 13x_4 + 19x_6 + 19x_9 + 21x_{10} + 8x_{11} + 29x_{14} + 11x_{15} + 25x_{16} + 3x_{17} \leq 22$

$C22: 26x_2 + 30x_3 + 19x_4 + 2x_{12} + 3x_{13} + 3x_{17} + 29x_{19} \leq 44$

$C23: 14x_1 + 3x_3 + 13x_4 + 10x_5 + 9x_8 + 19x_9 + 2x_{10} + 9x_{11} + 5x_{13} + 24x_{15} + 22x_{16} + 27x_{17} + 19x_{18}$
$+ 7x_{19} \leq 96$

$C24: 27x_2 + 26x_4 + 27x_5 + 15x_{10} + 21x_{13} + 4x_{14} + 16x_{15} + 25x_{19} \leq 49$

$C25: 15x_3 + 13x_4 + 10x_5 + 7x_7 + 24x_8 + 12x_9 + 10x_{11} + 28x_{13} + 18x_{14} + 10x_{15} + 10x_{17} + 14x_{19} \leq 42$

$C26: 16x_1 + 12x_4 + 13x_6 + 18x_7 + 3x_8 + x_9 + 4x_{10} + x_{13} + 28x_{15} + 24x_{16} + 19x_{17} + 9x_{20} \leq 54$

$C27: 15x_1 + 25x_2 + 12x_7 + 3x_9 + 2x_{13} + 21x_{14} + 27x_{15} + 23x_{17} \leq 21$

$C28: 20x_2 + 20x_3 + 16x_7 + 17x_8 + 26x_{11} + 24x_{12} + 10x_{18} \leq 53$

$C29: 22x_3 + 22x_6 + 25x_7 + 3x_8 + 21x_9 + x_{11} + 15x_{14} + 12x_{15} + 24x_{18} + 4x_{19} \leq 89$

$C30: 16x_1 + 27x_2 + 30x_5 + 9x_8 + 27x_{10} + 16x_{12} + 15x_{14} + 17x_{15} + 7x_{16} + 11x_{18} + 16x_{20} \leq 35$

Binaries

x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15} x_{16} x_{17} x_{18} x_{19} x_{20}

Appendix C: Instance generator

We generate a total of 900 instances for the computational study. From the total of 900 instances, 300 are pure binary instances (which will be used for both minimization and maximization forms), 300 are pure continuous instances for the minimization form, and 300 are pure continuous instances for the maximization form. We generate the binary instances in such a way that they become challenging to solve both in the form of minimization or maximization. Note that generating pure continuous instances using the same settings that we employ for generating binary instances often results in instances that can be solved in just a fraction of a second by GRB, and that cannot be used for making any meaningful comparisons. Hence, we make some changes in the settings when creating continuous instances. Moreover, as mentioned in the introduction, continuous instances in maximization form can be solved in polynomial time (while mMPs remain NP-hard even in pure continuous form). Therefore, our pure continuous instances in the maximization form are larger than our pure continuous instances in the minimization form.

Specifically, pure binary instances are divided into three classes based on their value of $p \in \{2,3,4\}$. Each class contains 20 subclasses of instances based on the dimensions of the matrix $A_{m\times n}$, and each subclass contains 5 randomly generated instances. We consider $n \in \{100,200,300,400,500\}$ and $m=\alpha n$, where $\alpha \in \{0.5,1,1.5,2\}$. For example, our smallest subclass is 100×50 with $n=100$ x-variables and $m=50$ constraints (related to x-variables), i.e., $\alpha=0.5$, and our largest subclass is 500×1000 with $n=500$ x-variables and $m=1000$ constraints (related to x-variables), i.e., $\alpha=2$. The sparsity of matrix A is set to 50%, i.e. $s_A := 0.5$, and the entries of matrix A are randomly drawn from the discrete uniform distribution $[1,30]$. The components of vector $\boldsymbol{b}$ are randomly drawn from the discrete uniform distribution $[ns_A, 10ns_A]$, where ns_A is the expected number of non-zero elements in each row of matrix A. The sparsity of matrix D is also set to 50%, and its entries in row $i \in I$ are drawn randomly from the discrete uniform distribution $[-10i, 10i]$. Note that each row represents a linear function defining a y-variable. By generating the entries of D in that way, different y-variables require different numbers of bits to be represented. To ensure that the instances are non-trivial in the form of both minimization and maximization, we make the y-variables highly conflicting. Specifically, if we decide to assign a non-zero value to the entry located in row $i \in I$ and column $j \in \{1,\ldots,n\}$ of matrix D, we first count how many times column j has taken positive and negative values in previously generated rows of D. If the number of positives (negatives) is larger than the number of negatives (positives), then we make sure that the value of the entry located in row $i \in I$ and column $j \in \{1,\ldots,n\}$ is negative (positive). In the case of a tie, no restriction on the sign is imposed. Finally, to assure that every objective function takes a positive value, we first assume that all the elements of vector $\boldsymbol{d}$ are zero and solve an optimization problem for each $i \in I$ to compute the minimum possible value for y_i, denoted by L_i. We then randomly draw the components of $\boldsymbol{d}$ from the discrete uniform distribution $[|L_i|+1, |L_i|+10]$.

For pure continuous instances in minimization form, we generate 300 instances similar to the procedure described for generating pure binary instances. The only difference is that the entries of matrix A for pure continuous instances are randomly drawn from the discrete uniform distribution $[-30, 30]$ (rather than $[1, 30]$). This makes the feasible set of the instances larger and consequently instances are expected to become more challenging to solve. We also generate 300 instances for the maximization form similarly to the procedure described for minimization. The only difference is that the dimensions of matrix $A_{m\times n}$ are set to larger values. Specifically, we use $n \in \{400, 800, 1200, 1600, 2000\}$ and $m = \alpha n$, where $\alpha \in \{0.5, 1, 1.5, 2\}$ for pure continuous maximization instances. This implies that the smallest subclass of the pure continuous instance in maximization form has 400 x-variables and 200 constraints (related to x-variables). Similarly, the largest subclass of the pure continuous instance in maximization form has 2000 x-variables and 4000 constraints (related to x-variables).

Appendix D: Detailed experimental results

In this section, we provide the details of all our four experiments outlined in Section 5. Throughout this section, we use two types of charts for comparing the performance of different algorithms/solvers, one for time comparison and the other for solution quality comparison.

For solution time comparisons, we use a specific *boxplot* in which on the horizontal axis different algorithms are listed, and on the vertical axis the run time ratio is provided. Specifically, for constructing a boxplot, for each instance and for each solution approach, we need to compute the ratio of the run time of the approach to the minimum of the run times of all approaches for that specific instance. Hence, the closer the ratio is to one, the better the approach. Note that we do not report the solutions times explicitly when describing the details of our experiments in this section. This is because interested readers may refer to the 185-page supplementary PDF document (or its corresponding CSV file) available at https://github.com/paymanghasemi/Multiplicative-Programs-by-Binary-encoding-the-Multiplication-Operation to see the details for every instance and run. However, for convenience, at the end of this section, we report the averages solution times (in seconds) of our best solution approaches compared to GRB SOCP or GRB Nonconvex for different subclasses of instances in the form of tables.

For solution quality comparisons, we employ *performance profiles* as they provide more details compared to a boxplot (Dolan and Moré 2002). The reason that we need to show more details to reader is that GRB SOCP or GRB Nonconvex can face numerical issues and report incorrect optimal solutions when solving a multiplicative program. We measure the solution quality of an approach for a multiplicative program as the gap between the objective value reported by the approach and the best objective value reported by all algorithms available on the same chart. We refer to this gap as the *best gap*, which helps identifying the cases where GRB SOCP or GRB Nonconvex face numerical issues.

In light of the above, the solution quality performance profiles present cumulative distribution functions for a set of solution approaches being compared with respect to their best gaps. The performance profiles show the best gaps on the horizontal axis and, on the vertical axis, for each approach, show the percentage of instances with a best gap that is smaller than or equal to the best gap on the horizontal axis. This means that values in the upper left-hand corner of the graph indicate the best performance.

Experiment 1: identifying the best search mechanism

In this experiment, we intend to find which of our proposed search mechanisms, including One-shot, Bitwise, Bitwise + P-cut, and Bitwise + F-cut, is the fastest in solving multiplicative programs. We limit the focus of this experiment to instances with $p = 2$ as the Altogether and Nested formulations are the same for such instances. Figure 3 shows the solution time comparison of different

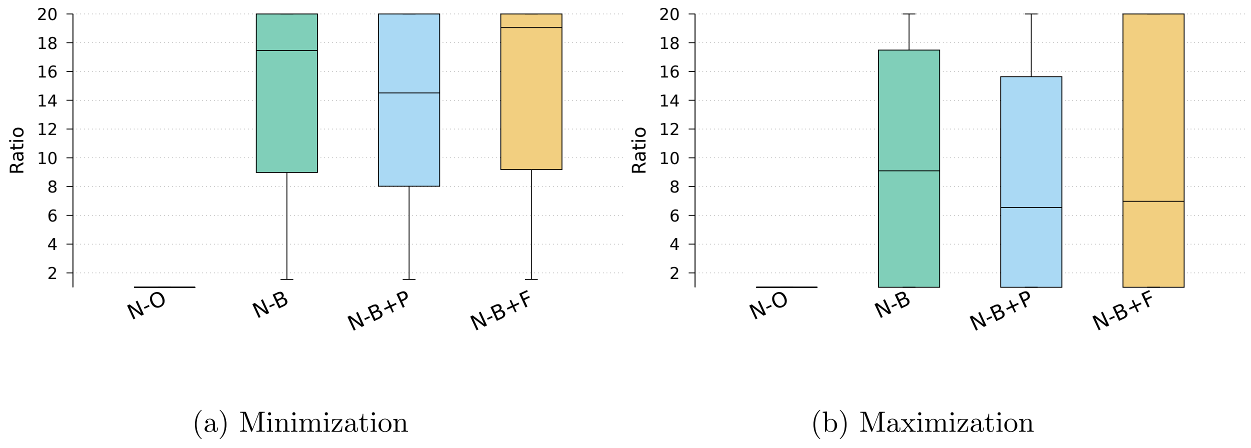


Figure 3 Solution time comparison between the proposed search mechanisms for pure binary instances with $p=2$

search mechanisms. From this figure, we can observe that for both minimization and maximization instances, One-shot is the fastest search mechanism. The median of the One shot approach solution times is 6 to 20 times smaller than the solution times of other search mechanisms. This is not surprising as Bitwise is more suitable for resolving the curse of multiplication (and not for instances with small values of p). In the remaining experiments, we use the One-shot search mechanism whenever calling our proposed method. As an aside, we note that from Figure 3, we also observe that Bitwise + P-cut performs significantly better than Bitwise and Bitwise + F-cut. Therefore, users may want to use Bitwise + P-cut if they want to employ the Bitwise search mechanism.

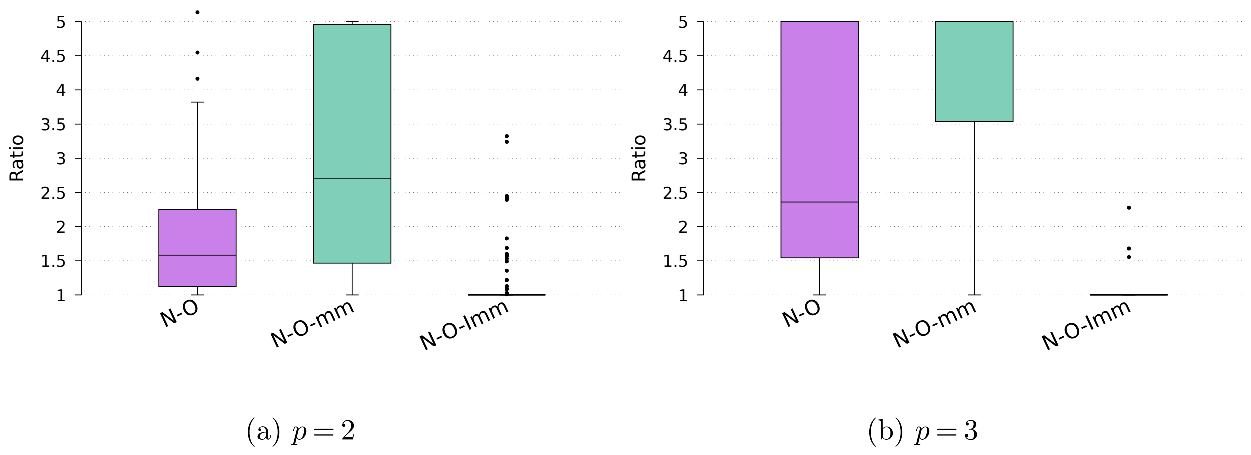
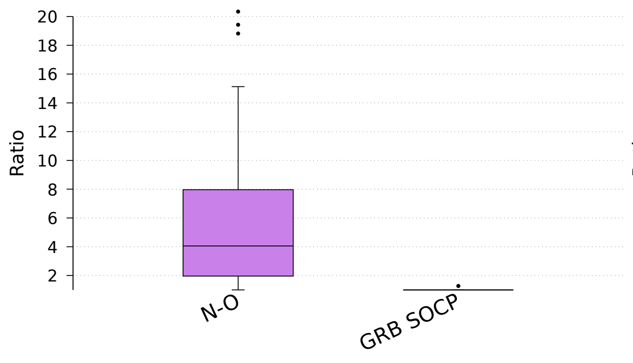


Figure 4 Solution time comparison between different enhancement settings for pure binary minimization instances

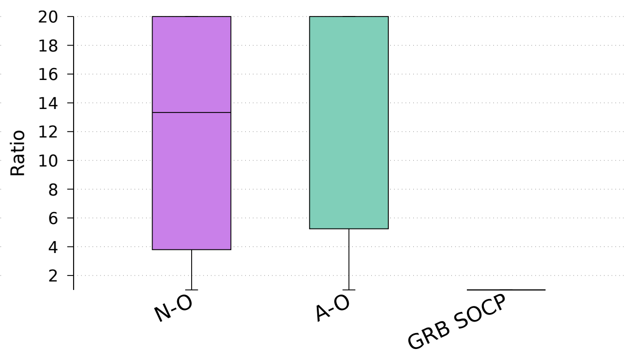
Experiment 2: identifying the best enhancement technique for minimization instances

By fixing One-shot as the default search mechanism (based on the results of Experiment 1), in this experiment, we test our proposed warm-start enhancements for minimum multiplicative programs.

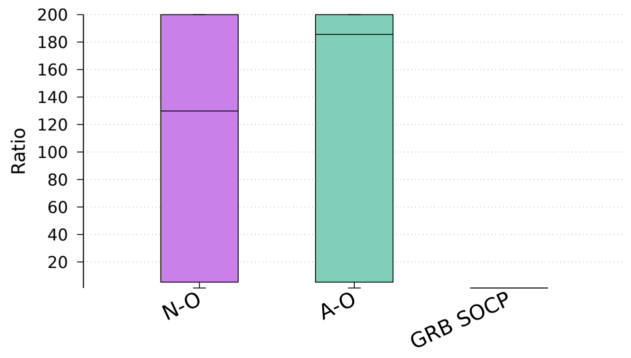
Figure 4 provides the solution time comparison between the three enhancement settings, i.e., no warm-start, Min-Min, and Indirect Min-Min, for pure binary multiplicative instances with $p \in \{2,3\}$. We later show in Experiment 3 that the Altogether reformulation does not perform well. Therefore, in this experiment, we use the Nested reformulation to deal with instances where $p = 3$. We can observe that Indirect Min-Min significantly decreases (by a factor of 1.5 to 5) the solution time for almost all instances. As an aside, we also observe that the Min-Min warm-start setting increases the overall solution time compared to using no warm-start as it is too time-consuming to solve the optimization problem for the Min-Min strategy. In the remaining experiments, the Indirect Min-Min setting will be active when solving minimization instances.



(a) $p = 2$



(b) $p = 3$

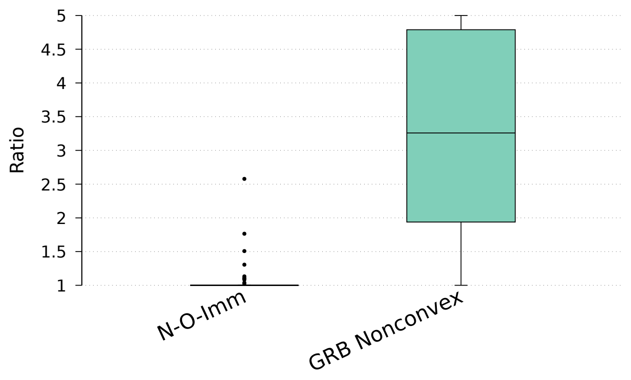


(c) $p = 4$

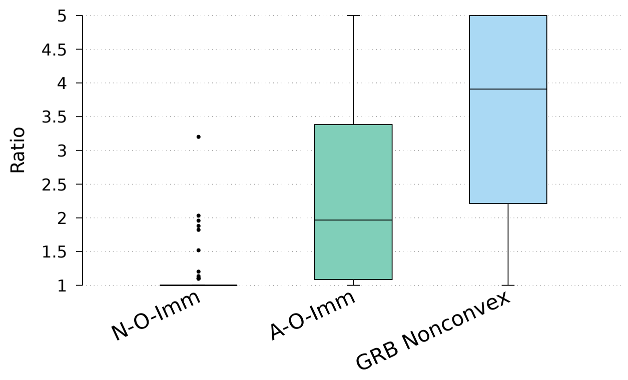
Figure 5 Solution time comparison between the proposed algorithms and GRB SOCP for binary maximization instances

Experiment 3: comparing different solution methods for binary instances

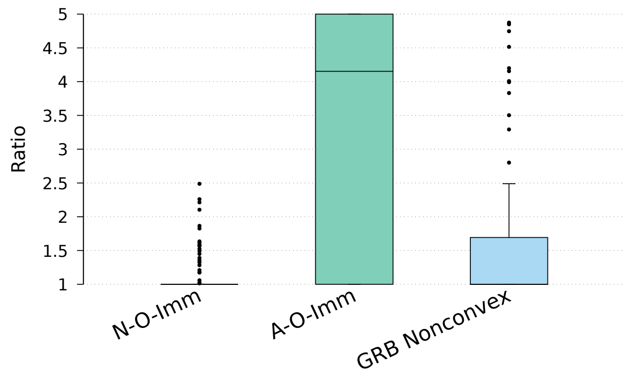
In this experiment, we compare the overall performance of our proposed solution approaches, including the Nested and Altogether reformulations (when One-shot and Indirect Min-Min are set as the default), with the performance of GRB SOCP and GRB Nonconvex on pure binary instances with $p \in \{2,3,4\}$. Figure 5 provides the solution time comparison on the pure binary maximum multiplicative instances. In Figure 5a, we can observe that GRB SOCP dominates our proposed solution approaches for any $p \in \{2,3,4\}$. Similarly, in Figures 5b and 5c, we can observe that the Nested reformulation performs better than the Altogether reformulation. However, it is evident that none of them are capable of competing with GRB SOCP.



(a) $p = 2$



(b) $p = 3$



(c) $p = 4$

Figure 6 Solution time comparison between the proposed algorithms and GRB Nonconvex for binary minimization instances

Figure 6 provides the solution time comparison on pure binary minimum multiplicative instances. We observe that our proposed Nested reformulation outperforms GRB Nonconvex for any $p \in \{2,3,4\}$. From Figures 6b and 6c, we observe that, for instances with $p \in \{3,4\}$, the Nested reformulation performs significantly better than the Altogether reformulation. Further, we observe that even the Altogether reformulation outperforms GRB Nonconvex on the binary minimum instances with $p = 3$. However, we observe from Figure 6c that GRB Nonconvex performs better than the

Altogether reformulation. Overall, due to the poor performance of the Altogether reformulation, we set the Nested reformulation as the default setting for the last experiment.

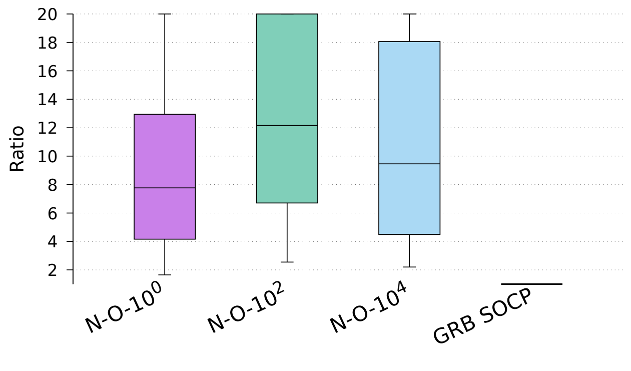
Experiment 4: comparing different solution methods for continuous instances

In this experiment, we compare the overall performance of our proposed Nested reformulation (when One-shot and Indirect Min-Min are set as the default) with the performance of GRB SOCP and GRB Nonconvex on pure continuous instances with $p \in \{2,3,4\}$. We note that for continuous instances, our approach can be used for only generating approximations (with any desirable level of precision) based on the discussion in Section 3.4. Hence, we conduct our experiments for three scenarios where 0, 2, and 4 digits after the decimal point are considered for each continuous y-variable, i.e., $\beta \in \{0,2,4\}$. In other words, we consider three multipliers $\{10^0, 10^2, 10^4\}$ for transforming the continuous y-variables into integers.

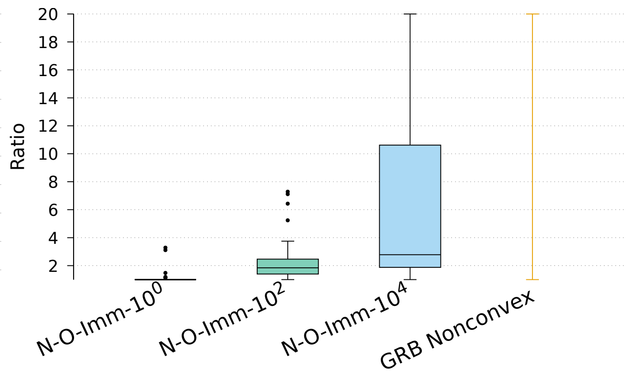
First, we report our results for only instances with $p = 2$. We note that for some continuous instances, GRB (including SOCP, Nonconvex, and its MIP solvers) reports errors, e.g., "unable to satisfy optimality tolerances", and/or terminates early. This was not observed for pure binary instances. While such issues can possibly be resolved by changing/tuning internal parameters of GRB (as described in its manual), we did not change them as it is not clear what the most efficient way of tuning is. Therefore, to have a fair comparison, we simply remove such instances in our comparisons in this section and report the number of removed instances wherever appropriate.

Figure 7 provides the performance comparison between the solution approaches for pure continuous instances with $p = 2$. From the continuous maximization instances with $p = 2$, a total of 16 instances are removed (due to errors or early termination) before generating the figure. However, no instances are removed for the minimization instances. For the maximization instances, we observe from Figure 7a that GRB SOCP dominates our proposed algorithm with respect to solution time. However, one interesting observation is that the solutions reported by our approach with multipliers of 10^2 and 10^4 have a better quality than those reported by GRB SOCP (although the optimality gap tolerance of GRB SOCP is set to zero). This again shows the power of the proposed solution method in handling numerical issues. For the maximization instances, we observe from Figure 7b that our proposed algorithm outperforms GRB Nonconvex with respect to solution time. However, we observe from Figure 7d that GRB Nonconvex has obtained better quality solutions. Overall, we observe that mMPs and MMPs have completely opposite performance compared to our proposed approach. One reason for this could be the fact that continuous MMPs are polynomially solvable while continuous mMPs are NP-hard.

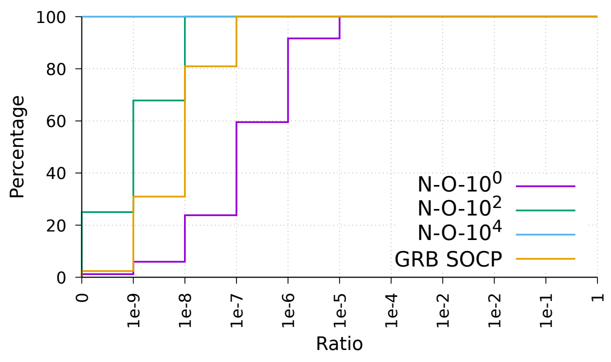
Figure 8 provides the performance comparison for pure continuous instances with $p = 3$. From the continuous maximization instances with $p = 3$, a total of 23 instances are removed (due to



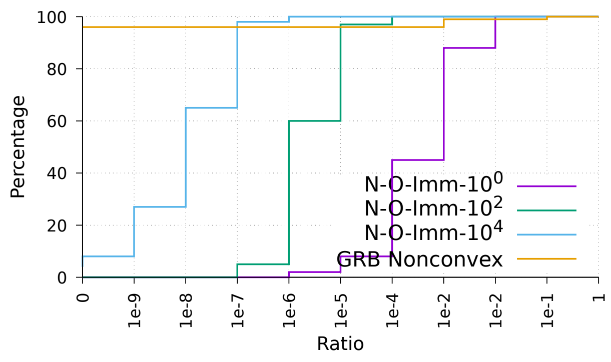
(a) Solution time comparison for maximization



(b) Solution time comparison for minimization



(c) Solution quality comparison for maximization

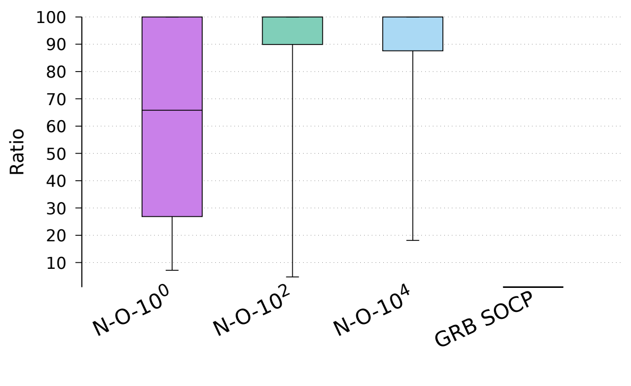


(d) Solution quality comparison for minimization

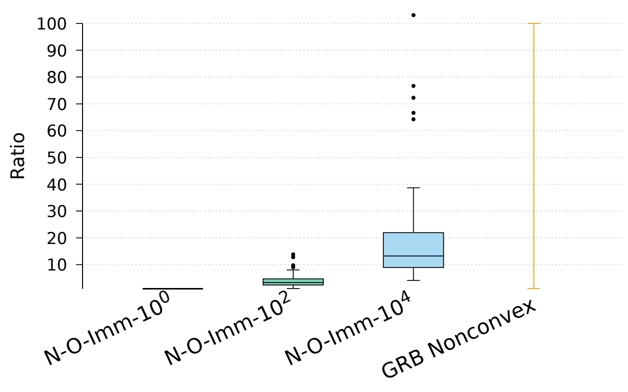
Figure 7 Performance comparison between the proposed approach, GRB SOCP, and GRB Nonconvex for pure continuous instances with $p = 2$

errors or early termination) before generating the figure. In total, only 2 instances are removed for the minimization instances. For the maximization instances, we observe from Figures 8a and 8c that GRB SOCP dominates our proposed algorithm with respect to both solution time and quality. However, for the minimization instances, we observe the opposite result. Specifically, Figures 8b and 8d show that our proposed algorithm outperforms GRB Nonconvex with respect to both solution time and quality (for the multiplier of 10^4).

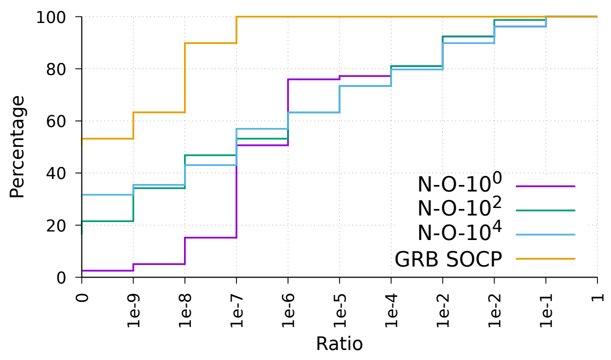
Figure 9 provides the performance comparison for pure continuous instances with $p = 4$. From the continuous maximization instances with $p = 4$, a total of 8 instances are removed (due to errors or early termination) before generating the figure. In total, 14 instances are removed for the minimization instances. For the maximization instances, we again observe from Figure 9a and 9c that GRB SOCP dominates our proposed algorithm with respect to both solution time and quality. However, for the minimization instances, we observe from Figures 9b and 9d that our proposed



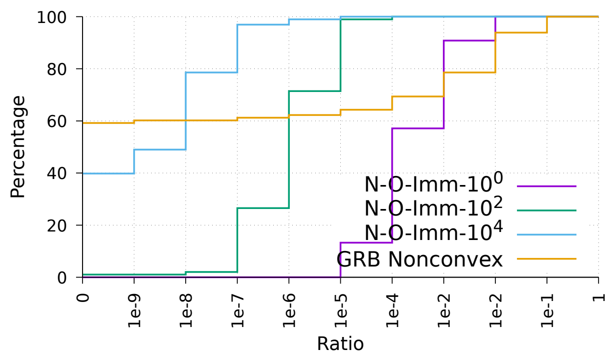
(a) Solution time comparison for maximization



(b) Solution time comparison for minimization



(c) Solution quality comparison for maximization



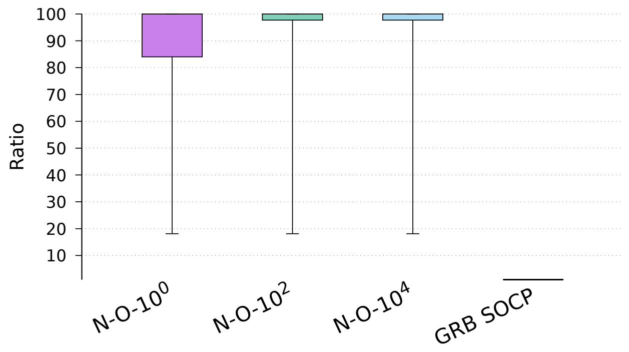
(d) Solution quality comparison for minimization

Figure 8 Performance comparison between the proposed approach, GRB SOCP, and GRB Nonconvex for pure continuous instances with $p = 3$

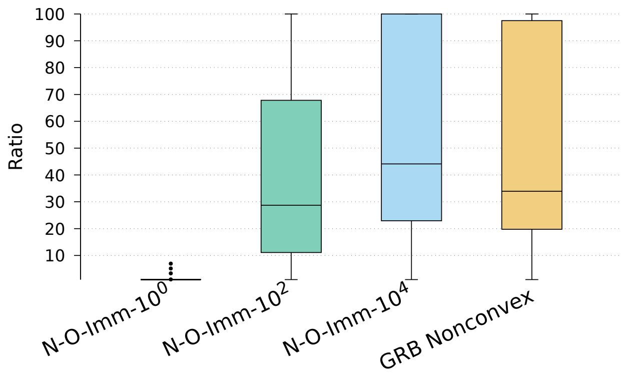
algorithm outperforms GRB Nonconvex with respect to both solution time and quality (for the multiplier of 10^0). We also observe that as the multiplier increases, we do not necessarily see an improvement in the quality of solutions reported. This is because when $p = 4$, the instances become larger and more difficult to solve. Hence, many instances with a multiplier of 10^4 cannot be solved to optimality within the time limit.

Report: average solution times

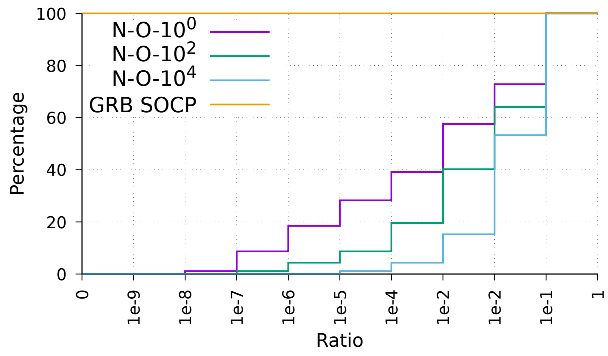
In the previous parts of the paper, we have not provided any information about the actual solution times (in seconds). So, the propose of this section is to report such information for interested readers. Note that based on Table 5, we know the best setting for our proposed solution approach for different types of instances. So, in this section, we report the average solution times (in seconds) per subclass for the best setting of our proposed solution approach compared to GRB (Nonconvex or SOCP). Specifically, Table 6 is for all binary instances (in both maximization and minimization



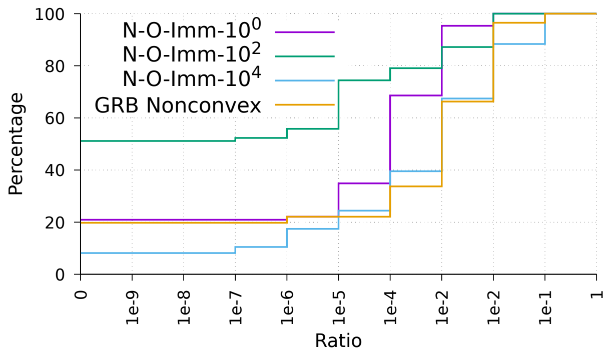
(a) Solution time comparison for maximization



(b) Solution time comparison for minimization



(c) Solution quality comparison for maximization



(d) Solution quality comparison for minimization

Figure 9 Performance comparison between the proposed approach, GRB SOCP, and GRB Nonconvex for pure continuous instances with $p = 4$

forms); Table 7 is for continuous instances in the form of minimization; Table 8 is for continuous instances in the form of minimization. The figures reported in each table are averages over 5 instances. Each figure reported in the row labeled 'Avg' shows the average of the four figures above it. Recall that the imposed time limit is 3600 seconds for each method and instance in our computational study. So, in the tables, 3600 means that the instance is not solved to optimality.

Table 6 Average solution time (sec.) per subclass of pure binary instances

Subclass	$p=2$				$p=3$				$p=4$			
	Max		Min		Max		Min		Max		Min	
($n \times m$)	GRB SOCP	N-O	GRB Nonconvex	N-O-Imm	GRB SOCP	N-O	GRB Nonconvex	N-O-Imm	GRB SOCP	N-O	GRB Nonconvex	N-O-Imm
100 × 50	0.14	0.54	0.25	0.10	0.45	3.55	1.53	0.65	0.29	77.13	11.39	9.25
100 × 100	0.25	0.60	0.36	0.16	0.48	4.33	3.38	0.91	0.37	71.80	10.39	11.33
100 × 150	0.36	0.61	0.49	0.23	0.70	8.93	4.19	0.97	0.44	63.32	17	19.01
100 × 200	0.76	1.01	0.88	0.30	1.11	11.11	4.59	0.98	0.77	41.61	14.52	19.73
Avg	**0.38**	**0.69**	**0.49**	**0.20**	**0.69**	**6.98**	**3.42**	**0.88**	**0.47**	**63.46**	**13.33**	**14.83**
200 × 100	1.54	7.38	1.14	0.36	3.01	88.74	11.47	2.61	2.77	2,870.68	132.07	95.27
200 × 200	2.01	6.94	3.84	1.09	3.44	87.61	32.67	22.14	6.35	2,381.17	223.37	197.34
200 × 300	3.69	11.86	5.73	2.11	6.18	173.19	56.30	20.44	8.13	1,967.04	469.03	272.54
200 × 400	7.61	21.29	14.78	15.95	24.75	704.89	140.78	63.46	4.79	1,480.42	597.52	369.64
Avg	**3.71**	**11.87**	**6.37**	**4.88**	**9.34**	**263.61**	**60.30**	**27.16**	**5.51**	**2,174.83**	**355.50**	**233.70**
300 × 150	5.96	36.52	12.73	11.22	132.87	1,973.46	111.50	134.90	21.37	3,600	695.87	669.48
300 × 300	51.35	252.81	76.21	20.03	43.78	1,793.30	364.95	179.79	31.95	3,600	2,978.88	1,198.98
300 × 450	141.68	916.02	62.26	26.45	42.69	1,885.81	683.88	182.68	68.78	3,600	3,548.87	1,906.17
300 × 600	294.83	1,576.01	308.89	79.05	784.29	2,171.42	1,320.08	367.16	80.69	3,600	3,600	3,304.56
Avg	**123.45**	**695.34**	**115.02**	**34.19**	**250.91**	**1,956**	**620.10**	**216.14**	**50.70**	**3,600**	**2,705.91**	**1,769.80**
400 × 200	45.21	917.19	38.47	6.03	377.79	3,049.79	406.28	164.66	24.35	3,600	2,374.11	1,080.32
400 × 400	831.15	2,189.04	272.19	80.87	2,299.85	3,600	1,596.66	564.67	1,931.86	3,600	3,600	3,039.43
400 × 600	1,582.25	3,600	599.28	426.08	506.79	3,355.18	3,045.58	524.68	1,588.82	3,600	3,600	2,923.85
400 × 800	805.74	2,517.37	994.61	486.01	2,195.25	3,600	3,127.87	593.89	814.04	3,600	3,600	3,600
Avg	**816.09**	**2,305.90**	**476.14**	**249.75**	**1,344.92**	**3,401.24**	**2,044.10**	**461.98**	**1,089.77**	**3,600**	**3,293.53**	**2,660.90**
500 × 250	620.12	2,559.68	851.29	195.43	1,598.27	3,600	1,311.07	163.22	1,040.80	3,600	3,600	3,600
500 × 500	2,948.09	3,600	1,251.10	506.82	3,238.07	3,600	3,142.02	1,886.16	2,338.58	3,600	3,600	3,137.05
500 × 750	2,281.89	3,249.34	1,765.53	652.18	2,937.46	3,600	3,600	3,600	2,950.63	3,600	3,600	3,600
500 × 1000	3,600	3,600	2,565.95	816.81	3,013.91	3,600	3,600	3,600	3,140.03	3,600	3,600	3,600
Avg	**2,362.52**	**3,252.25**	**1,608.47**	**542.81**	**2,696.93**	**3,600**	**2,913.27**	**2,312.35**	**2,367.51**	**3,600**	**3,600**	**3,484.26**

Table 7 Average solution time (sec.) per subclass of continuous minimization instances

Subclass	$p=2$		$p=3$		$p=4$	
$(n \times m)$	GRB Nonconvex	N-O-Imm-10^0	GRB Nonconvex	N-O-Imm-10^0	GRB Nonconvex	N-O-Imm-10^0
100 × 50	37.33	0.23	220.88	1.45	998.77	44.78
100 × 100	5.69	0.26	68.38	1.23	2,261.40	19.21
100 × 150	5.07	0.31	72.50	1.08	1,430.73	26.11
100 × 200	5.55	0.37	118.20	1.66	620.05	10.75
Avg	**13.41**	**0.29**	**119.99**	**1.35**	**1,327.74**	**25.21**
200 × 100	49.02	0.50	1,337.04	2.71	2,944.63	385.93
200 × 200	19.88	0.55	607.64	2.97	3,040.51	18.96
200 × 300	26.85	0.88	813.28	3.45	3,600	53.41
200 × 400	32.72	1.21	882.82	3.85	3,600	38.48
Avg	**32.12**	**0.79**	**910.20**	**3.25**	**3,296.28**	**124.19**
300 × 150	92.34	1.13	3,590.85	5.05	1,479.46	92.66
300 × 300	62.85	1.50	2,849.71	5.55	3,600	37.87
300 × 450	82.94	1.87	3,363.13	9	3,600	60.60
300 × 600	88.51	3.18	3,329.60	8.75	3,600	277.01
Avg	**81.66**	**1.92**	**3,283.32**	**7.09**	**3,069.87**	**117.04**
400 × 200	1,359.81	1.89	3,600	8.17	2,123.66	122
400 × 400	184.88	2.57	3,131.84	14.08	3,600	112.85
400 × 600	212.07	4.73	3,600	12.77	3,600	122.28
400 × 800	234.05	4.14	3,600	15.61	3,600	630.45
Avg	**497.70**	**3.33**	**3,482.96**	**12.66**	**3,230.92**	**246.90**
500 × 250	3,600	2.56	3,600	12.61	3,485.64	231.24
500 × 500	256.90	4.94	3,600	17.63	3,600	840.50
500 × 750	331.01	6.06	3,600	26.87	2,885.45	492.56
500 × 1000	308.96	8.09	3,600	27.51	3,600	218.81
Avg	**1,124.22**	**5.41**	**3,600**	**21.16**	**3,392.77**	**445.78**

Table 8 Average solution time (sec.) per subclass of continuous maximization instances

Subclass	$p=2$		$p=3$		$p=4$	
$(n\times m)$	GRB SOCP	N-O-10^0	GRB SOCP	N-O-10^0	GRB SOCP	N-O-10^0
400 × 200	0.14	2.78	0.15	19.78	0.10	2,724.88
400 × 400	0.37	4.20	0.40	286.35	0.28	3,192.79
400 × 600	0.68	4.17	0.75	190.12	0.55	3,513.46
400 × 800	1.22	7.16	1.28	93.40	0.93	3,347.84
Avg	**0.60**	**4.58**	**0.64**	**147.41**	**0.46**	**3,194.74**
800 × 400	0.69	16.45	0.75	96.25	0.53	3,443.42
800 × 800	2.28	29.45	2.51	755.75	1.86	3,600
800 × 1600	9.43	25.07	10.78	506.14	8.04	3,600
800 × 1200	5.19	50.11	5.71	141.58	4.33	3,600
Avg	**4.39**	**30.27**	**4.94**	**374.93**	**3.69**	**3,560.85**
1200 × 600	2.13	38.30	2.58	1,429.86	1.84	3,600
1200 × 1200	8.73	44.51	9.40	1,566.67	8.26	2,925.10
1200 × 1800	19.97	141.72	22.20	2,324.84	22.43	3,600
1200 × 2400	38.44	160.13	40.59	2,374.15	35.69	3,600
Avg	**17.32**	**96.16**	**18.69**	**1,923.88**	**17.05**	**3,431.27**
1600 × 800	6.56	51.67	7.13	2,069.30	5.56	3,600
1600 × 1600	23.81	228.91	25.65	1,428.67	23.85	3,600
1600 × 2400	52.18	283.11	53.67	2,611.75	56.41	3,600
1600 × 3200	88.48	354.25	100.31	2,259.04	94.07	3,600
Avg	**42.76**	**229.49**	**46.69**	**2,092.19**	**44.97**	**3,600**
2000 × 1000	13.21	169.92	15.94	3,598.26	14	3,600
2000 × 2000	44.85	367.85	52.53	3,600	56.13	3,600
2000 × 3000	104.11	662.59	117.80	2,983.96	114.12	3,600
2000 × 4000	176.46	987.35	192.10	2,872.06	196.33	3,600
Avg	**84.66**	**546.93**	**94.59**	**3,263.57**	**95.15**	**3,600**