A Branch-and-Bound Algorithm for a Class of Mixed Integer Linear Maximum Multiplicative Programs: A Multi-objective Optimization Approach

Payman Ghasemi Saghand, Hadi Charkhgard*, Changhyun Kwon

Department of Industrial and Management Systems Engineering, University of South Florida, USA

Abstract

We present a linear programming based branch-and-bound algorithm for a class of mixed integer optimization problems with a bi-linear objective function and linear constraints. This class of optimization problems can be viewed as a special case of the problem of optimization over the set of efficient solutions in multi-objective optimization. It is known that when there exists no integer decision variable, such a problem can be solved in polynomial time. In fact, in such a case, the problem can be transformed into a Second-Order Cone Program (SOCP) and so it can be solved efficiently by a commercial solver such as CPLEX SOCP solver. However, in a recent study, it is shown that such a problem can be solved even faster in practice by using a bi-objective linear programming based algorithm. So, in this study, we embed that algorithm in an effective branch-and-bound framework to solve mixed integer instances. We also develop several enhancement techniques including preprocessing and cuts. An extensive computational study demonstrate that the proposed branch-and-bound algorithm outperforms a commercial mixed integer SOCP solver. Moreover, the effect of different branching and node selecting strategies is explored.

Keywords: multiplicative programming, multi-objective optimization, optimization over the efficient set, linear programming, branch-and-bound algorithm

1. Introduction

Many real-world optimization problems involve multiple objectives. Such an optimization problem (with p objectives) can be stated as follows:

$$\max_{\boldsymbol{x}\in\mathcal{X}} \{z_1(\boldsymbol{x}),\dots,z_p(\boldsymbol{x})\}, \tag{1}$$

where $\mathcal{X}\subseteq\mathbb{R}^n$ represents the set of feasible solutions of the problem. The objectives of such a problem are often conflicting, i.e., it is impossible to find a feasible solution that simultaneously optimizes all objectives. Consequently, it is not surprising that the focus of multi-objective optimization community has been primarily on developing effective techniques for generating some, if not all, *efficient* solutions, i.e., solutions in which it is impossible to improve the value of one objective without a deterioration in the value of at least one other objective of multi-objective

*Corresponding author
Email address: `hcharkhgard@usf.edu` (Hadi Charkhgard)

optimization problems. In fact, in recent years, significant advances have been made on developing exact solution approaches for multi-objective optimization problems, see for instance Boland et al. (2017b); Dächert and Klamroth (2014); Kirlik and Sayın (2014); Lokman and Köksalan (2013); Özlen et al. (2013); Özpeynirci and Köksalan (2010); Przybylski and Gandibleux (2017); Soylu and Yıldız (2016); Ehrgott and Gandibleux (2007); and Eusébio et al. (2014). This is mainly because understanding the trade-offs between objectives can help decision makers select their preferred solutions.

Although understanding the trade-offs between objectives can be valuable, some researchers argue (see for instance Jorge (2009); and Boland et al. (2017a)) that presenting too many efficient solutions can confuse a decision maker, and so may make selecting a preferred solution almost impossible. An approach that alleviates this issue is finding a preferred solution among the set of efficient solutions directly. This approach is known as *optimizing over the efficient set*, which is a global optimization problem (Benson, 1984). Let $\mathcal{X}_E$ be the set of efficient solutions of Problem (1). Also, let $f(\boldsymbol{x})$ be a function representing how decision makers will choose their preferred solutions from $\mathcal{X}_E$. The problem of optimizing over the efficient set can be stated as follows:

$$\max_{\boldsymbol{x}\in\mathcal{X}_E} f(\boldsymbol{x}).$$

A specific subclass of such optimization problems is *maximum multiplicative programs*, i.e., problems of the form:

$$\max_{\boldsymbol{x}\in\mathcal{X}} \prod_{i=1}^{p} z_i(\boldsymbol{x}).$$

where $z_i(\boldsymbol{x}) \geq 0$ for all $\boldsymbol{x} \in \mathcal{X}$ and $i = 1, \dots, p$. It is known (see for example Nash, 1950) that a maximum multiplicative program is equivalent to the following problem:

$$\max_{\boldsymbol{x}\in\mathcal{X}_E} \prod_{i=1}^{p} z_i(\boldsymbol{x}).$$

Therefore, a maximum multiplicative program is a special case of the problem of optimization over the efficient set since we can set $f(\boldsymbol{x}) = \prod_{i=1}^{p} z_i(\boldsymbol{x})$. Overall, there are several studies in the literature about applications of *linear* maximum multiplicative programs, i.e., all variables are continuous, $\mathcal{X}$ is defined by a set of linear constraints, and $z_i(\boldsymbol{x})$ is linear for all $i = 1, \dots, p$. This is mainly because such optimization problems often arise in game theory settings where the $\boldsymbol{x}$ variables represent players' actions and $\boldsymbol{z}(\boldsymbol{x}) := \big(z_1(\boldsymbol{x}), \dots, z_p(\boldsymbol{x})\big)$ represents players' utilities. Examples include computing the Nash solution to a bargaining problem (Nash, 1950, 1953), computing an equilibrium of a linear Fisher or a Kelly capacity allocation market (Eisenberg and Gale, 1959; Chakrabarty et al., 2006; Jain and Vazirani, 2007; Vazirani, 2012a,b). Moreover, this class of optimization problems has applications in system reliability as well (Ardakan et al., 2016; Feizabadi and Jahromi, 2017; Zhang and Chen, 2016).

A convex programming solver, for example one that uses an interior point method, can find an optimal solution to a linear maximum multiplicative program in polynomial time (Grötschel et al., 1988). Note that since we assume that the optimal objective value of a linear maximum multiplicative program is strictly positive, it is possible to use $\sum_{i=1}^{p} \log y_i$ as the objective function. This transformation can help us to solve a linear maximum multiplicative program faster in practice. In fact, Charkhgard et al. (2018) have computationally shown that even a faster way of solving linear maximum multiplicative program is to reformulate such a problem as a Second-Order Cone Program (SOCP) using the technique developed by Ben-Tal and Nemirovski (2001). This is mainly because such a reformulation enables us to use the power of commercial SOCP solvers, e.g., CPLEX SOCP, for solving linear maximum multiplicative programs.

Some authors (see for instance Vazirani, 2012b) argue that because convex programming solvers are significantly slower than linear programming solvers, it may be possible to develop even faster solvers for linear maximum multiplicative programs. In order to do so, one should develop an exact algorithm that can solve a linear maximum multiplicative program by solving a number of (single-objective) linear programs. Using this observation and the fact that linear maximum multiplicative programs are special cases of the problem of optimization over the efficient set of multi-objective linear programs, Charkhgard et al. (2018) were able to develop a linear programming based algorithm to solve a maximum multiplicative programs when $p = 2$ in polynomial time. It is numerically shown that this algorithm outperforms CPLEX SOCP solver significantly, i.e., by a factor of up to three, for large-sized instances.

As an aside, we note that linear maximum multiplicative programs appear to be closely related to linear *minimum* multiplicative programs (see for instance Gao et al., 2006; Ryoo and Sahinidis, 2003; Shao and Ehrgott, 2014; and Shao and Ehrgott, 2016). Specifically, by changing the objective function of a linear maximum multiplicative programs from *max* to *min* a linear minimum multiplicative program is obtained. However, it is known that a linear minimum multiplicative program is NP-hard (Shao and Ehrgott, 2016). Therefore, because of this significant difference, we do not consider this class of optimization problems in this study.

In light of the above, the main contribution of our research is extending the algorithm proposed by Charkhgard et al. (2018) to solve any mixed integer linear maximum multiplicative program with $p = 2$, i.e., those that some of their decision variables have to take integer values. We propose an effective branch-and-bound algorithm which employs the power the algorithm developed by Charkhgard et al. (2018) for solving mixed integer instances. We also develop several preprocessing and cut generating techniques to improve the solution time of the proposed algorithm. Moreover, the effects of several branching and node selecting strategies are explored. An extensive computational study show that, for large-sized mixed binary instances, our proposed approach outperforms a commercial solver, i.e., CPLEX Mixed Integer SOCP solver, by a factor of around two on average. Also, for general mixed integer instances, our proposed algorithm outperforms CPLEX Mixed Integer SOCP solver not only on large-sized instances but also on small-sized instances by a factor of two on average.

The remainder of the paper is organized as follows. In Section 2, we provide some preliminaries and explain a high-level description of the algorithm proposed by Charkhgard et al. (2018). In Section 3, we explain our proposed branch-and-bound algorithm in detail. In Section 4, some branching strategies are introduced. In Section 5, some node selecting strategies are presented. In Section 6, we explain some potential enhancement techniques. In Section 7, we conduct an extensive computational study. Finally, in Section 8, we give some concluding remarks.

2. Preliminaries

A Mixed Integer Linear Maximum Multiplicative Program with $p = 2$ can be stated as follows:

$$
\begin{aligned}
\max\ & \prod_{i=1}^{2} y_i \\
\text{s.t. } & \boldsymbol{y} = D\boldsymbol{x} + \boldsymbol{d} \\
& A\boldsymbol{x} \leq \boldsymbol{b} \\
& \boldsymbol{x}, \boldsymbol{y} \geq \boldsymbol{0}, \quad \boldsymbol{x} \in \mathbb{R}^{n_c} \times \mathbb{B}^{n_b} \times \mathbb{Z}^{n_i}, \quad \boldsymbol{y} \in \mathbb{R}^2,
\end{aligned}
\tag{2}
$$

where n_c, n_b, and n_i represent the number of continuous, binary, and integer decision variables, respectively. Also, D is a $2 \times n$ matrix where $n := n_c + n_b + n_i$, $\boldsymbol{d}$ is a 2-vector, A is an $m \times n$ matrix, and $\boldsymbol{b}$ is an m-vector.

The focus of this study is on solving Problem (2). We refer to the set $\mathcal{X} := \{\boldsymbol{x} \in \mathbb{R}^n : A\boldsymbol{x} \leq \boldsymbol{b},\ \boldsymbol{x} \geq \boldsymbol{0}\}$ as *the feasible set in the decision space* and to the set $\mathcal{Y} := \{\boldsymbol{y} \in \mathbb{R}^2 : \boldsymbol{x} \in \mathcal{X},\ \boldsymbol{y} = D\boldsymbol{x} + \boldsymbol{d},\ \boldsymbol{y} \geq \boldsymbol{0}\}$ as *the feasible set in the criterion space*. We assume that $\mathcal{X}$ is bounded (which implies that $\mathcal{Y}$ is compact) and the optimal objective value of the problem is strictly positive, i.e., there exists a $\boldsymbol{y} \in \mathcal{Y}$ such that $\boldsymbol{y} > \boldsymbol{0}$. We usually refer to $\boldsymbol{x} \in \mathcal{X}$ as a *feasible solution* and to $\boldsymbol{y} \in \mathcal{Y}$ as a *feasible point* ($\boldsymbol{y}$ is the image of $\boldsymbol{x}$ in the criterion space).

Definition 1. *A feasible solution $\boldsymbol{x} \in \mathcal{X}$ is called efficient, if there is no other $\boldsymbol{x}' \in \mathcal{X}$ such that $y_1 \leq y'_1$ and $y_2 < y'_2$ or $y_1 < y'_1$ and $y_2 \leq y'_2$ where $\boldsymbol{y} := D\boldsymbol{x} + \boldsymbol{d}$ and $\boldsymbol{y}' := D\boldsymbol{x}' + \boldsymbol{d}$. If $\boldsymbol{x}$ is efficient, then $\boldsymbol{y}$ is called a nondominated point. The set of all efficient solutions is denoted by $\mathcal{X}_E$. The set of all nondominated points is denoted by $\mathcal{Y}_N$ and referred to as the nondominated frontier.*

Proposition 1. *An optimal solution of Problem (2), denoted by $\boldsymbol{x}^*$, is an efficient solution and therefore its corresponding image in the criterion space, denoted by $\boldsymbol{y}^*$ where $\boldsymbol{y}^* := D\boldsymbol{x}^* + \boldsymbol{d}$, is a nondominated point.*

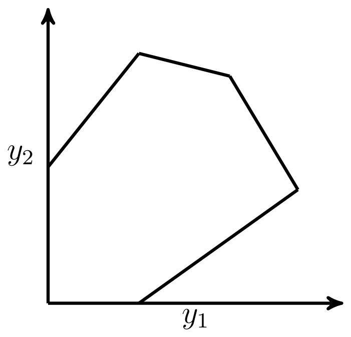
PROOF. Suppose that $\boldsymbol{x}^*$ is an optimal solution of Problem (2) but it is not an efficient solution. By definition, this implies that there must exist a feasible solution denoted by $\boldsymbol{x} \in \mathcal{X}$ that dominates $\boldsymbol{x}^*$. In other words, we must have that either $y^*_1 \leq y_1$ and $y^*_2 < y_2$ or $y^*_1 < y_1$ and $y^*_2 \leq y_2$ (where $\boldsymbol{y} := D\boldsymbol{x} + \boldsymbol{d}$). Also, by assumptions of Problem (2) we know that $\boldsymbol{y}^* > \boldsymbol{0}$. Therefore, we must have that $0 < y^*_1 y^*_2 < y_1 y_2$. Consequently, $\boldsymbol{x}^*$ cannot be an optimal solution (a contradiction). $\square$

Proposition 1 implies that Problem (2) is equivalent to $\max_{\boldsymbol{y} \in \mathcal{Y}_N} y_1 y_2$ and this is precisely optimization over the efficient set. Charkhgard et al. (2018) used this observation and developed an algorithm, which we refer to as CST (which comes from the names of its authors Charkhgard, Savelsbergh, and Talebian) in this study, to solve a relaxation of Problem (2). The relaxation can be obtained by dropping the integrality condition of binary and integer decision variables of Problem (2).

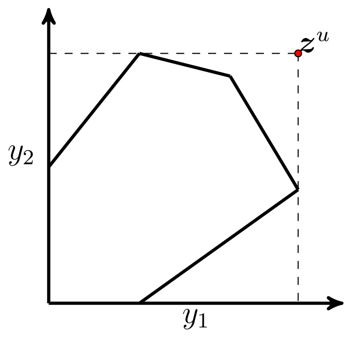
By definition, the optimal objective value of such a relaxation should provide a dual bound, i.e, an upper bound, for the optimal objective value of Problem (2). Therefore, by this observation, in this paper, we frequently use CST to compute dual bounds. However, whenever we use CST, we provide a lower bound vector, denoted by $\boldsymbol{l} \in \mathbb{R}^n$, and an upper bound vector, denoted by $\boldsymbol{u} \in \mathbb{R}^n$, for $\boldsymbol{x}$. As an aside, we note that $\boldsymbol{l}$ and $\boldsymbol{u}$ will be provided/updated automatically during the course of the branch-and-bound algorithm that we will introduce in the next section. So, we introduce the operation CST$(\boldsymbol{l}, \boldsymbol{u})$ which is equivalent to solving the following problem:

$$\begin{aligned}
\max\ & \prod_{i=1}^{2} y_i \\
\text{s.t. } & \boldsymbol{y} = D\boldsymbol{x} + \boldsymbol{d} \\
& A\boldsymbol{x} \leq \boldsymbol{b} \\
& \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u} \\
& \boldsymbol{x}, \boldsymbol{y} \geq \boldsymbol{0}, \quad \boldsymbol{x} \in \mathbb{R}^n, \quad \boldsymbol{y} \in \mathbb{R}^2,
\end{aligned} \tag{3}$$

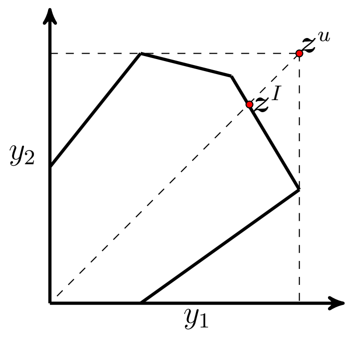
by calling CST algorithm. This operation simply returns an optimal solution and an optimal point of Problem (3) denoted by $(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{y}})$. Note that if $\tilde{\boldsymbol{x}}$ =null (or equivalently $\tilde{\boldsymbol{y}}$=null) then Problem (3) is infeasible. Next, we provide a high-level description of the CST algorithm. Interested readers can refer to Charkhgard et al. (2018) for further details.



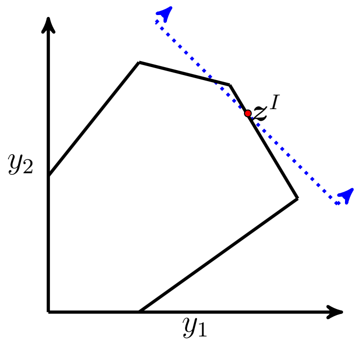
(a) The feasible set in the criterion space



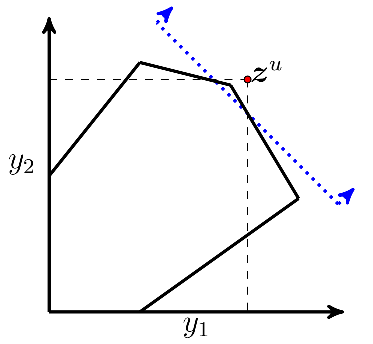
(b) The first upper bound point



(c) The first intersection point



(d) The first cut



(e) The second upper bound point

Figure 1: An illustration of the workings of CST$(\boldsymbol{l}, \boldsymbol{u})$

Suppose that CST$(\boldsymbol{l}, \boldsymbol{u})$ is called. An illustration of the feasible set of Problem (3) in the criterion space can be found in Figure 1a. In each iteration, CST computes a global upper bound point, denoted by $\boldsymbol{y}^u$, and a global lower bound point, denoted by $\boldsymbol{y}^l$, in the criterion space.

This implies that $\boldsymbol{y}_1^u\boldsymbol{y}_2^u$ and $\boldsymbol{y}_1^l\boldsymbol{y}_2^l$ provide a global upper bound and a global lower bound for the optimal objective value of Problem (3), respectively. The algorithm terminates whenever the (relative and/or absolute) optimality gap falls below a given threshold. In the first iteration, the algorithm computes an upper bound point by solving two single-objective linear programs. The first maximizes y_1 and the second maximizes y_2 over the feasible set in the criterion space. An illustration of the upper bound point in the criterion space can be found in Figure 1b. Next, the algorithm searches over the imaginary line the passes through the upper bound point and the origin to compute the so-called intersection point which is denoted by $\boldsymbol{y}^I$. The intersection point is the closest feasible point in the criterion space to the upper bound point and can be computed by solving a single-objective linear program. Therefore, $y_1^I y_2^I$ provides by definition a global lower bound for the optimal objective value of Problem (3). An illustration of the intersection point found in the first iteration can be found in Figure 1c. Note that the algorithm keeps the best global lower bound found during the course of the algorithm in $\boldsymbol{y}^l$. In other words, after computing an intersection point, $\boldsymbol{y}^l$ should be updated. Next, the algorithm adds a cut to the criterion space based on the position of the intersection point to remove the parts of the criterion space that cannot contain a better lower bound point. An illustration of the cut added in the first iteration can be found in Figure 1d. After adding the cut, the next iteration starts and the same process repeats but this time over the reduced feasible set in the criterion space. For example, an illustration of the upper bound point produced in the second iteration can be found in Figure 1e.

3. A branch-and-bound algorithm

Our proposed algorithm is similar to the traditional branch-and-bound algorithm for single-objective mixed integer linear programs. The main difference is that instead of using a linear programming solver to compute dual bounds, we employ the CST algorithm. Next, we provide a detailed explanation of our proposed branch-and-bound algorithm.

The algorithm maintains a queue of nodes, denoted by TREE. The algorithm also maintains a global upper bound, denoted by $\mathrm{G_{UB}}$, and global lower bound, denoted by $\mathrm{G_{LB}}$. At the beginning, the algorithm sets $\mathrm{G_{UB}} = +\infty$ and $\mathrm{G_{LB}} = -\infty$. Moreover, the algorithm initializes $(\boldsymbol{l}, \boldsymbol{u})$ with $(\boldsymbol{0}, +\infty)$. To initialize the queue, the algorithm first calls $\mathrm{CST}(\boldsymbol{l}, \boldsymbol{u})$ to compute $(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{y}})$. If the integrality conditions hold, i.e., $\tilde{\boldsymbol{x}} \in \mathbb{R}^{n_c} \times \mathbb{B}^{n_b} \times \mathbb{Z}^{n_i}$, then an optimal solution is found. So, in that case, the algorithm terminates after setting $(\boldsymbol{x}^*, \boldsymbol{y}^*) = (\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{y}})$. Otherwise, by definition, $\mathrm{CST}(\boldsymbol{l}, \boldsymbol{u})$ has computed a dual bound, and so the algorithm sets $\mathrm{G_{UB}} = \tilde{y}_1\tilde{y}_2$ and initializes the queue by $(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{y}}, \boldsymbol{l}, \boldsymbol{u})$. The algorithm then explores the queue as long as it is nonempty and $\mathrm{G_{UB}} - \mathrm{G_{LB}} \geq \varepsilon_1$ and $\frac{\mathrm{G_{UB}} - \mathrm{G_{LB}}}{\mathrm{G_{UB}}} \geq \varepsilon_2$, where $\varepsilon_1, \varepsilon_2 \in (0, 1)$ are the user-defined absolute and relative optimality gap tolerances, respectively. Next, we explain how each element of the queue is explored.

In each iteration, the algorithm pops out an element of the queue and denote it by $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{l}, \boldsymbol{u})$. Note that when an element is popped out from the queue then that element does not exist in the queue anymore. We will explain in detail in which order the elements of the queue should be popped out in Section 5. The algorithm next selects an index of a decision variable that was supposed to take an integer (or binary) value but it currently has fractional value in solution $\boldsymbol{x}$. This operation is denoted by BRANCHING-INDEX$(\boldsymbol{x})$ and its output is denoted by j. In Section 4, we will introduce several rules for this operation.

Next, the algorithm generates two new lower bound and upper bound vectors, denoted by $(\boldsymbol{l}^1, \boldsymbol{u}^1)$ and $(\boldsymbol{l}^2, \boldsymbol{u}^2)$. The algorithm sets $l_i^1 = l_i$ for all $i \in \{1, \dots, n\} \backslash \{j\}$, $l_j^1 = \lceil x_j \rceil$, and $\boldsymbol{u}^1 = \boldsymbol{u}$. Similarly, the algorithm sets $u_i^2 = u_i$ for all $i \in \{1, \dots, n\} \backslash \{j\}$, $u_j^2 = \lfloor x_j \rfloor$, and $\boldsymbol{l}^2 = \boldsymbol{l}$. The

Algorithm 1: A Branch-and-Bound Algorithm

```
Input: A feasible instance of Problem (2)
$Queue.create(\textsc{Tree})$
$(\boldsymbol{l}, \boldsymbol{u}) \leftarrow (\boldsymbol{0}, +\boldsymbol{\infty})$
$\mathrm{G}_{\mathrm{LB}} \leftarrow -\infty$; $\mathrm{G}_{\mathrm{UB}} \leftarrow +\infty$
$(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{y}}) \leftarrow \mathrm{CST}(\boldsymbol{l}, \boldsymbol{u})$
if $\tilde{\boldsymbol{x}} \in \mathbb{R}^{n_c} \times \mathbb{B}^{n_b} \times \mathbb{Z}^{n_i}$ then
    $(\boldsymbol{x}^*, \boldsymbol{y}^*) \leftarrow (\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{y}})$
    $\mathrm{G}_{\mathrm{LB}} \leftarrow \tilde{y}_1 \tilde{y}_2$
else
    $\textsc{Tree}.add\big((\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{y}}, \boldsymbol{l}, \boldsymbol{u})\big)$
    $\mathrm{G}_{\mathrm{UB}} \leftarrow \tilde{y}_1 \tilde{y}_2$
while not $Queue.empty(\textsc{Tree})$ & $\mathrm{G}_{\mathrm{UB}} - \mathrm{G}_{\mathrm{LB}} \geq \varepsilon_1$ & $\frac{\mathrm{G}_{\mathrm{UB}} - \mathrm{G}_{\mathrm{LB}}}{\mathrm{G}_{\mathrm{UB}}} \geq \varepsilon_2$ do
    $\textsc{Tree}.PopOut\big((\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{l}, \boldsymbol{u})\big)$
    $j \leftarrow \textsc{Branching-Index}(\boldsymbol{x})$
    $(\boldsymbol{l}^1, \boldsymbol{u}^1) \leftarrow (\boldsymbol{l}, \boldsymbol{u})$; $l_j^1 \leftarrow \lceil x_j \rceil$
    $(\boldsymbol{l}^2, \boldsymbol{u}^2) \leftarrow (\boldsymbol{l}, \boldsymbol{u})$; $u_j^2 \leftarrow \lfloor x_j \rfloor$
    $(\boldsymbol{x}^1, \boldsymbol{y}^1) \leftarrow \mathrm{CST}(\boldsymbol{l}^1, \boldsymbol{u}^1)$
    if $(\boldsymbol{x}^1, \boldsymbol{y}^1) \neq$ (null,null) then
        if $\boldsymbol{x}^1 \in \mathbb{R}^{n_c} \times \mathbb{B}^{n_b} \times \mathbb{Z}^{n_i}$ & $y_1^1 y_2^1 > \mathrm{G}_{\mathrm{LB}}$ then
            $(\boldsymbol{x}^*, \boldsymbol{y}^*) \leftarrow (\boldsymbol{x}^1, \boldsymbol{y}^1)$
            $\mathrm{G}_{\mathrm{LB}} \leftarrow y_1^1 y_2^1$
        else if $y_1^1 y_2^1 - \mathrm{G}_{\mathrm{LB}} \geq \varepsilon_1$ & $\frac{y_1^1 y_2^1 - \mathrm{G}_{\mathrm{LB}}}{y_1^1 y_2^1} \geq \varepsilon_2$ then
            $\textsc{Tree}.add\big((\boldsymbol{x}^1, \boldsymbol{y}^1, \boldsymbol{l}^1, \boldsymbol{u}^1)\big)$
    $(\boldsymbol{x}^2, \boldsymbol{y}^2) \leftarrow \mathrm{CST}(\boldsymbol{l}^2, \boldsymbol{u}^2)$
    if $(\boldsymbol{x}^2, \boldsymbol{y}^2) \neq$ (null,null) then
        if $\boldsymbol{x}^2 \in \mathbb{R}^{n_c} \times \mathbb{B}^{n_b} \times \mathbb{Z}^{n_i}$ & $y_1^2 y_2^2 > \mathrm{G}_{\mathrm{LB}}$ then
            $(\boldsymbol{x}^*, \boldsymbol{y}^*) \leftarrow (\boldsymbol{x}^2, \boldsymbol{y}^2)$
            $\mathrm{G}_{\mathrm{LB}} \leftarrow y_1^2 y_2^2$
        else if $y_2^2 y_2^2 - \mathrm{G}_{\mathrm{LB}} \geq \varepsilon_1$ & $\frac{y_1^2 y_2^2 - \mathrm{G}_{\mathrm{LB}}}{y_1^2 y_2^2} \geq \varepsilon_2$ then
            $\textsc{Tree}.add\big((\boldsymbol{x}^2, \boldsymbol{y}^2, \boldsymbol{l}^2, \boldsymbol{u}^2)\big)$
    Update $\mathrm{G}_{\mathrm{UB}}$
return $\boldsymbol{x}^*, \boldsymbol{y}^*$
```

algorithm first explores $(\boldsymbol{l}^1, \boldsymbol{u}^1)$. This implies that the algorithm calls $\text{CST}(\boldsymbol{l}^1, \boldsymbol{u}^1)$ to compute $(\boldsymbol{x}^1, \boldsymbol{y}^1)$. Again we note that if CST fails to find a feasible solution then we have that $\boldsymbol{x}^1 =$null and $\boldsymbol{y}^1 =$null. Therefore, in that case, the algorithm will skip the following steps and proceed to explore $(\boldsymbol{l}^2, \boldsymbol{u}^2)$. Otherwise, the algorithm first checks whether the integrality conditions hold, i.e., $\boldsymbol{x}^1 \in \mathbb{R}^{n_c} \times \mathbb{B}^{n_b} \times \mathbb{Z}^{n_i}$, and $\text{G}_{\text{LB}} < y_1^1 y_2^1$. If that is the case then a new and better global lower bound is found and so the algorithm will set $(\boldsymbol{x}^*, \boldsymbol{y}^*) = (\tilde{\boldsymbol{x}}^1, \tilde{\boldsymbol{y}}^1)$ and $\text{G}_{\text{LB}} = y_1^1 y_2^1$. Otherwise, the algorithm checks whether $y_1^1 y_2^1 - \text{G}_{\text{LB}} \geq \varepsilon_1$ and $\frac{y_1^1 y_2^1 - \text{G}_{\text{LB}}}{y_1^1 y_2^1} \geq \varepsilon_2$. If that is the case then the algorithm will add $(\boldsymbol{x}^1, \boldsymbol{y}^1, \boldsymbol{l}^1, \boldsymbol{u}^1)$ to be explored further in the future because it is possible to find better feasible solutions by exploring that element.

After exploring $(\boldsymbol{l}^1, \boldsymbol{u}^1)$, $(\boldsymbol{l}^2, \boldsymbol{u}^2)$ should be explored similarly. Therefore, the algorithm will explore it and then before starting the next iteration it will update G_{UB}. The maximum value of $y_1 y_2$ among all nodes in the queue defines the new global upper bound. A detailed description of the proposed branch-and-bound algorithm can be found in Algorithm 1.

4. Branching strategies

Selecting a branching variable is a crucial task in any branch-and-bound algorithm since it can impact the solution time of the algorithm significantly. Ideally, we would like to select a variable for branching that helps us to explore the minimum number of nodes in total. In light of this observation, in this study, we explore some variants of several well-known branching strategies that will define the operation $\text{Branching-Index}(\boldsymbol{x})$ in Algorithm 1. Let I be the index sets of all binary and integer decision variables in Problem (2). For a given node of the queue $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{l}, \boldsymbol{u})$, let $V(\boldsymbol{x}) \subseteq I$ be the index set of all variables in solution $\boldsymbol{x}$ that were supposed to take integer (or binary) values but they have taken fractional values. We have explored the following five branching strategies in this study (interested readers can refer to Bonami et al., 2011 for further details):

- **Random branching**: This is the easiest and the least-memory consuming branching rule. In this rule, for a given node of the queue $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{l}, \boldsymbol{u})$, all the elements of $V(\boldsymbol{x})$ have an equal chance of being selected, and so the algorithm selects one of them randomly.

- **Most infeasible branching**: For a given node of the queue $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{l}, \boldsymbol{u})$, another simple branching strategy is to randomly select an element of $V(\boldsymbol{x})$ with the largest integer violation for branching. For each $i \in V(\boldsymbol{x})$, the integer violation is defined as $\min(x_i - \lfloor x_i \rfloor, \lceil x_i \rceil - x_i)$.

  In the traditional branch-and-bound algorithm for single-objective mixed integer linear programming, it is known that this branching rule does not outperform random branching (Achterberg et al., 2005). In fact, this rule normally has a poor performance. However, surprisingly, we will computationally show (in Section 7) that this rule performs the best for our proposed branch-and-bound algorithm for solving Problem (2).

- **Pseudo-costs branching**: This strategy was initially introduced by Bénichou et al. (1971) and then explored further by Martin (1999). This strategy maintains a history of the results of past branchings for each $i \in I$. For a given node of the queue $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{l}, \boldsymbol{u})$, the algorithm selects an index $i \in V(\boldsymbol{x})$ with the largest expected change in the dual bound, i.e., upper bound. Next, we explain how pseudo-costs can be computed for each iteration.

  Let ρ_i^1 and ρ_i^2 be the pseudo-costs for $i \in I$ at any time during the course of the algorithm. For each $i \in I$, at the beginning, the algorithm sets $\rho_i^1 = 0$ and $\rho_i^2 = 0$. An any iteration of the algorithm, if the algorithm branches on $i \in I$ and calls $\text{CST}(\boldsymbol{l}^1, \boldsymbol{u}^1)$, the algorithm updates ρ_i^1 as follows:

$$\rho_i^1 \leftarrow \rho_i^1 + \frac{y_1 y_2 - y_1^1 y_2^1}{\lceil x_i \rceil - x_i},$$

where $y_1 y_2$ is the optimal objective value associated with the node that the algorithm is exploring at that iteration, i.e., $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{l}, \boldsymbol{u})$. We denote the number of times that ρ_i^1 has been updated by n_i^1 at any time during the course of the algorithm. Note that if after calling $\text{CST}(\boldsymbol{l}^1, \boldsymbol{u}^1)$ it turns out that $(\boldsymbol{x}^1, \boldsymbol{y}^1)$ =(null,null), i.e., meaning the problem is infeasible, then ρ_i^1 and n_i^1 should not be updated. Similarly, whenever the algorithm branches on $i \in I$ and calls $\text{CST}(\boldsymbol{l}^2, \boldsymbol{u}^2)$, the algorithm updates ρ_i^2 as follows:

$$\rho_i^2 \leftarrow \rho_i^2 + \frac{y_1 y_2 - y_1^2 y_2^2}{x_i - \lfloor x_i \rfloor}.$$

We denote the number of times that ρ_i^2 has been updated by n_i^2 at any time during the course of the algorithm. Note that if after calling $\text{CST}(\boldsymbol{l}^2, \boldsymbol{u}^2)$ it turns out that $(\boldsymbol{x}^2, \boldsymbol{y}^2)$ =(null,null), i.e., meaning the problem is infeasible, then ρ_i^2 and n_i^2 should not be updated.

In light of the above, suppose that at a particular iteration of the algorithm, we are exploring the node $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{l}, \boldsymbol{u})$ and we want to decide on which variable we should branch. Based on the pseudo-costs, one can estimate the change that can occur in the dual bound by calling $\text{CST}(\boldsymbol{l}^1, \boldsymbol{u}^1)$ after branching on $i \in V(\boldsymbol{x})$ as follows:

$$\delta_i^1 := (\lceil x_i \rceil - x_i) \frac{p_i^1}{n_i^1}.$$

In other words, we estimate that $y_1 y_2 - y_1^1 y_2^1 \approx \delta_i^1$. Similarly, one can estimate the change that can occur in the dual bound by calling $\text{CST}(\boldsymbol{l}^2, \boldsymbol{u}^2)$ after branching on $i \in V(\boldsymbol{x})$ as follows:

$$\delta_i^2 := (x_i - \lfloor x_i \rfloor) \frac{p_i^2}{n_i^2}$$

In other words, we estimate that $y_1 y_2 - y_1^2 y_2^2 \approx \delta_i^2$. Therefore, the algorithm can compute a score for each $i \in V(\boldsymbol{x})$ as follows:

$$s_i := \mu \min(\delta_i^1, \delta_i^2) + (1 - \mu) \max(\delta_i^1, \delta_i^2)$$

where $\mu \in [0, 1]$ is a user-defined parameter which is typically close to 1. In this paper, we set $\mu = 0.9$ since it computationally performs the best. The index with the highest score is the one that the algorithm selects for branching. In order to employ this sophisticated rule, an initial estimation is required. In this paper, we use the following strategies to obtain an initial estimation:

1. **Initialization using the random branching**: In this strategy, we employ random branching for a number of iterations (which is 10 in this paper) and then the algorithm switches to pseudo-costs branching. It is worth mentioning that if at a particular iteration the pseudo-costs branching fails to select an index, i.e., $s_i = 0$ for all $i \in V(\boldsymbol{x})$, then we employ the random branching for that particular iteration.

2. **Initialization using the most infeasible branching**: This is similar to the previous case, we only employ the most infeasible branching instead of the random branching.

- **Reliability branching:** This branching is mostly based on the *strong* branching (ILOG, 2003). Suppose that at a particular iteration of the algorithm, we are exploring the node $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{l}, \boldsymbol{u})$ and we want to decide on which variable we should branch using the strong branching. In strong branching, the algorithm will actually call both $\mathrm{CST}(\boldsymbol{l}^1, \boldsymbol{u}^1)$ and $\mathrm{CST}(\boldsymbol{l}^2, \boldsymbol{u}^2)$ to compute the exact change in the dual bound, i.e., $\Delta_i^1 := y_1 y_2 - y_1^1 y_2^1$ and $\Delta_i^2 := y_1 y_2 - y_1^2 y_2^2$. So, for each $i \in V(\boldsymbol{x})$, the algorithm computes s_i but using Δ_i^1 and Δ_i^2 instead of δ_i^1 and δ_i^2. Similar to the pseudo-costs branching, the index with the highest score is the one that the algorithm selects for branching. Since strong branching imposes a high computational burden, Achterberg et al. (2005) suggest that we should combine the strong branching with the pseudocosts branching. This is known as the *reliability branching.* If $n_i^1 \leq \tau$ then we use Δ_i^1 in computing s_i, and otherwise we use δ_i^1. Also, if $n_i^2 \leq \tau$ then we use Δ_i^2 in computing s_i, and otherwise we use δ_i^2. In this paper, it is assumed that $\tau = 2$ since computationally it performs the best.

5. Node selecting strategies

Selecting which node should be popped out in each iteration is another crucial task in any branch-and-bound algorithm since it can impact the solution time of the algorithm significantly. The aim of node selecting strategies is pruning open nodes and ending the queue as quickly as possible. This can be done by either finding a good feasible solution, in order to increase the global lower bound (or primal bound), or decreasing the global upper bound (or dual bound). We implemented five different node selecting strategies, two of which are solely based on pseudocosts branching (Goux and Leyffer, 2002; Belotti et al., 2013):

- **Depth-first search**: In this strategy, the most recent added node to the tree is chosen for branching (Little et al., 1963). The two advantages of this strategy are that, if there is no primary feasible solution on hand, this strategy finds one quickly, and by doing so, it increases $\mathrm{G_{LB}}$ quickly. The other advantage is that it keeps the list of open nodes minimal, and therefore, the memory-usage is little. However, on the negative side, this strategy is slow in improving the global dual bound and so it takes a lot of time to prove the optimality of a solution.

- **Best-bound search**: This strategy selects the node with the best upper/dual bound (Land and Doig, 1960). Although this strategy needs more memory in comparison to other strategies, if an optimal solution is available, this strategy is the fastest to prove its optimality. Another important characteristic of this strategy is that, in practice, the first feasible solution that the algorithm finds under this strategy is usually an optimal solution of the problem.

- **Two-phase method**: The idea of this strategy is to combine depth-first search and best-bound search strategies to use the benefits of both. In this paper, our proposed algorithm employs the depth-first search strategy for at most $\min(1000, 3n)$ number of iterations. As soon as a feasible solution is found which is better than the current $\mathrm{G_{LB}}$ or the algorithm reaches to the maximum number of iterations then it switches to the best-bound search. The algorithm operates on the best-bound search for $\min(1000, 3n)$ number of iterations. As soon as the algorithm reaches to its maximum number of iterations then the algorithm checks how much the global upper bound has improved during operating on the best-bound search strategy. If the improvement is greater than or equal to 5% then the algorithm starts to operate on the best-bound search for $\min(1000, 3n)$ number of iterations again. Otherwise, it will return to the depth-first search strategy and similar procedures discussed above will be repeated.

- **Best expected bound**: This strategy is based on the pseudo-costs and selects the node with the best expected dual bound after branching. For any given node $N := (\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{l}, \boldsymbol{u})$ of the queue, let $\delta^1(N)$ and $\delta^2(N)$ be the change estimate in the dual bound computed by the pseudo-costs branching if we branch on the variable that pseudo-costs branching suggests for that particular node. So, in that case, the estimated dual bounds will be for node $N \in \textsc{Tree}$:

$$\bar{e}_N^1 := y_1 y_2 - \delta^1(N),$$

and

$$\bar{e}_N^2 := y_1 y_2 - \delta^2(N).$$

In light of the above, in this strategy, the algorithm selects a node $N \in \textsc{Tree}$ that has the largest value of $\max\{\bar{e}_N^1, \bar{e}_N^2\}$.

- **Best estimate**: This strategy is also based on the pseudo-costs. This strategy selects a node that is expected to result in the best integer solution (or best primal bound). The best expected primal bound for a given node $N := (\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{l}, \boldsymbol{u})$ of the queue can be computed as follows:

$$\underline{e}_N := y_1 y_2 - \sum_{i \in V(\boldsymbol{x})} \min\{\delta_i^1, \delta_i^2\}$$

In light of the above, in this strategy, the algorithm selects a node $N \in \textsc{Tree}$ that has the largest value of $\underline{e}_N$.

6. Enhancements

In this section, we explain a preprocessing technique which is developed for the aim of producing good global lower bounds and cuts for Problem (2). The preprocessing technique can be called before running Algorithm 1 to possibility improve the performance of this algorithm. Obviously, generating a good global lower/primal bound can be helpful since the nodes of the branch-and-bound tree can be pruned faster. Also, by generating cuts and adding them to the formulation, better global upper/dual bounds can be computed during the course of Algorithm 1.

The proposed preprocessing technique is developed based on Proposition 1. By this proposition, any efficient solution is expected to be a high-quality feasible solution and hence it can be considered as a (good) global lower bound for the problem. Therefore, in the proposed preprocessing technique, we attempt to enumerate some of the efficient solutions of the problem. In order to do so, we use the weighted sum operation, denoted by $\text{WSO}(\lambda_1, \lambda_2)$:

$$(\bar{\boldsymbol{x}}, \bar{\boldsymbol{y}}) \in \arg\max \{\lambda_1 y_1 + \lambda_2 y_2 : \boldsymbol{y} = D\boldsymbol{x} + \boldsymbol{d},\ A\boldsymbol{x} \le \boldsymbol{b},\ \boldsymbol{x} \ge \boldsymbol{0},\ \boldsymbol{y} \ge \boldsymbol{0},\ \boldsymbol{x} \in \mathbb{R}^{n_c} \times \mathbb{B}^{n_b} \times \mathbb{Z}^{n_i},\ \boldsymbol{y} \in \mathbb{R}^2\},$$

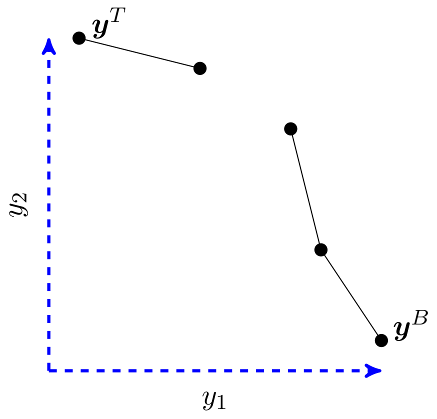


Figure 2: An illustration of the nondominated frontier corresponding to Problem (2)

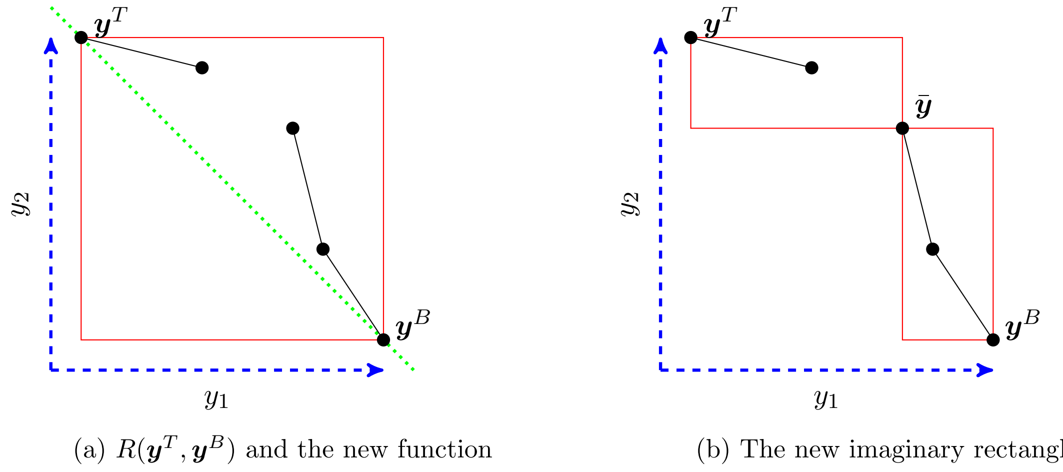


Figure 3: An illustration of the workings of the weighted sum method

where $\lambda_1, \lambda_2 > 0$ are user-defined weights. In multi-objective optimization, it is known that this operation always returns an efficient solution (if exists any) for a multi-objective optimization problem. However, for non-convex optimization problems such as Problem (2), not all efficient solutions can be computed using this operation by considering even all possible weights (Ehrgott, 2005). In addition to the fact that $\bar{y}_1\bar{y}_2$ provides a global lower bound, the weighted sum operation can naturally produce the following cut for Problem (1), due to optimality:

$$\lambda_1 y_1 + \lambda_2 y_2 \leq \lambda_1 \bar{y}_1 + \lambda_2 \bar{y}_2.$$

In light of the above, in the proposed preprocessing technique, we employ the weighted sum operation to generate a feasible solution (i.e., global primal bound) and a cut for the problem. In order to use the wighted sum operation effectively, we employ the *Weighted Sum Method (WSM)* (Aneja and Nair, 1979). This is because the WSM can compute all nondominated points that are extreme points of the convex hull of $\mathcal{Y}$. Next, we explain a high-level description of the WSM.

In the WSM, we first compute the endpoints of the nondominated frontier. The top endpoint, denoted by $\boldsymbol{y}^T$, can be computed by first solving,

$$(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{y}}) \in \arg\max \{y_2 : \\ \boldsymbol{y} = D\boldsymbol{x} + \boldsymbol{d},\ A\boldsymbol{x} \leq \boldsymbol{b},\ \boldsymbol{x} \geq \boldsymbol{0},\ \boldsymbol{y} \geq \boldsymbol{0},\ \boldsymbol{x} \in \mathbb{R}^{n_c} \times \mathbb{B}^{n_b} \times \mathbb{Z}^{n_i},\ \boldsymbol{y} \in \mathbb{R}^2\},$$

and if it is feasible, it needs to be followed by solving

$$(\boldsymbol{x}^T, \boldsymbol{y}^T) \in \arg\max \{y_1 :\ y_2 \geq \tilde{y}_2 \\ \boldsymbol{y} = D\boldsymbol{x} + \boldsymbol{d},\ A\boldsymbol{x} \leq \boldsymbol{b},\ \boldsymbol{x} \geq \boldsymbol{0},\ \boldsymbol{y} \geq \boldsymbol{0},\ \boldsymbol{x} \in \mathbb{R}^{n_c} \times \mathbb{B}^{n_b} \times \mathbb{Z}^{n_i},\ \boldsymbol{y} \in \mathbb{R}^2\},$$

We denote the operation of computing $(\boldsymbol{x}^T, \boldsymbol{y}^T)$ by $\arg\operatorname{lex}\max(y_2, y_1)$. Similarly, the bottom endpoint, can be computed by using $\arg\operatorname{lex}\max(y_1, y_2)$ in which its outcome is denoted by $(\boldsymbol{x}^B, \boldsymbol{y}^B)$. An illustration of the nondominated frontier corresponding to Problem (1) can be found in Figure 2. It is evident that if $(\boldsymbol{x}^T, \boldsymbol{y}^T) =$(null,null), i.e., Problem (2) is infeasible, then we must have that $(\boldsymbol{x}^B, \boldsymbol{y}^B) =$(null,null). Therefore, there is no need to compute $(\boldsymbol{x}^B, \boldsymbol{y}^B)$ in that case. Also, if $\boldsymbol{y}^T = \boldsymbol{y}^B$ then the Problem (2) has an *ideal* point, i.e., $|\mathcal{Y}_N| = 1$. In the remaining, we assume that $\boldsymbol{y}^T \neq \boldsymbol{y}^B$.

At the beginning of the WSM, we set $\mathrm{G_{LB}} = -\infty$. After computing the endpoints of the nondominated frontier, both $y_1^T y_2^T$ and $y_1^B y_2^B$ are global lower bounds for Problem (1). So, we set $\mathrm{G_{LB}}$ to the best/highest lower bound found and update $(\boldsymbol{x}^*, \boldsymbol{y}^*)$ accordingly. Moreover, we know that $y_1 \leq y_1^B$ and $y_2 \leq y_2^T$ are two valid cuts that can be added to the formulation and so we add them to the list of cuts which is denoted by Cuts. Note that after computing the endpoints, all other nondominated points must be in the imaginary rectangle defined by $\boldsymbol{y}^T$ and $\boldsymbol{y}^B$, denoted by $R(\boldsymbol{y}^T, \boldsymbol{y}^B)$ (See Figure 3a). Hence, the WSM explores this rectangle and may change it into smaller rectangles, and repeat this process in each one until it finds all extreme nondominated points.

More precisely, to explore a given rectangle $R(\boldsymbol{y}^1, \boldsymbol{y}^2)$ with $y_1^1 < y_1^2$ and $y_2^1 > y_2^2$, the WSM calls $\mathrm{WSO}(\lambda_1, \lambda_2)$ after setting $\lambda_1 = y_2^1 - y_2^2$ and $\lambda_2 = y_1^2 - y_1^1$ to compute $(\bar{\boldsymbol{x}}, \bar{\boldsymbol{y}})$. Note that in this way, $\lambda_1 y_1 + \lambda_2 y_2$ is a function which is parallel to the line that connects y^1 and y^2 in the criterion space. Figure 3a shows an example with $y^1 = y^T$ and $y^2 = y^B$. As discussed before since $\bar{\boldsymbol{y}}$ is a nondominated point, if $\mathrm{G_{LB}} < \bar{y}_1 \bar{y}_2$ then we set $\mathrm{G_{LB}} = \bar{y}_1 \bar{y}_2$ and $(\boldsymbol{x}^*, \boldsymbol{y}^*) = (\bar{\boldsymbol{x}}, \bar{\boldsymbol{y}})$. Also, the WSM adds $\lambda_1 y_1 + \lambda_2 y_2 \leq \lambda_1 \bar{y}_1 + \lambda_2 \bar{y}_2$ to the list of cuts. Finally, it is evident that if $\lambda_1 \bar{y}_1 + \lambda_2 \bar{y}_2 > \lambda_1 y_1^1 + \lambda_2 y_2^1$ then $\bar{\boldsymbol{y}}$ is not a convex combination of $\boldsymbol{y}^1$ and $\boldsymbol{y}^2$. So, in that case, a similar procedure is applied recursively to search $R(\boldsymbol{y}^1, \bar{\boldsymbol{y}})$ and $R(\bar{\boldsymbol{y}}, \boldsymbol{y}^2)$ for additional nondominated points (see Figure 3b).

Algorithm 2 shows a precise description of the weighted sum method or the proposed preprocessing technique.

7. Computational study

As we will explain later in this section, based on the study of Ben-Tal and Nemirovski (2001), Problem (2) can be reformulated as a mixed integer SOCP. So, in this section, we compare the performance of our proposed branch-and-bound algorithm with the performance of the mixed integer SOCP solver of CPLEX 12.7 (CPLEX-MI-SOCP). We implement our algorithm in C++ and use CPLEX 12.7 to solve linear programs and mixed integer programs arising during the course of Algorithm 1 and Algorithm 2. The computational experiments are conducted on a Dell PowerEdge

Algorithm 2: Preprocessing

```
Input: An instance of Problem (2)
$List.create(\textsc{Cuts})$
$Queue.create(Q)$
$\mathrm{G}_{\mathrm{LB}} = -\infty$
$(\boldsymbol{x}^T, \boldsymbol{y}^T) \leftarrow \arg\operatorname{lex}\max(y_2, y_1)$
if $(\boldsymbol{x}^T, \boldsymbol{y}^T) \neq (null, null)$ then
    $\boldsymbol{y}^B \leftarrow \arg\operatorname{lex}\max(y_1, y_2)$
    $\mathrm{G}_{\mathrm{LB}} \leftarrow y_1^T y_2^T$, $(\boldsymbol{x}^*, \boldsymbol{y}^*) \leftarrow (\boldsymbol{x}^T, \boldsymbol{y}^T)$
    $\textsc{Cuts}.add(y_2 \leq y_2^T)$
    if $\boldsymbol{y}^T \neq \boldsymbol{y}^B$ then
        $Q.add(\boldsymbol{R}(\boldsymbol{y}^T, \boldsymbol{y}^B))$
        if $\mathrm{G}_{\mathrm{LB}} < y_1^B y_2^B$ then
            $\mathrm{G}_{\mathrm{LB}} \leftarrow y_1^B y_2^B$, $(\boldsymbol{x}^*, \boldsymbol{y}^*) \leftarrow (\boldsymbol{x}^B, \boldsymbol{y}^B)$
        $\textsc{Cuts}.add(y_1 \leq y_1^B)$
while $not\ Queue.empty(\textsc{Tree})$ do
    $Q.PopOut(\boldsymbol{R}(\boldsymbol{y}^1, \boldsymbol{y}^2))$
    $\lambda_1 \leftarrow y_2^1 - y_2^2$
    $\lambda_2 \leftarrow y_1^2 - y_1^1$
    $(\bar{\boldsymbol{x}}, \bar{\boldsymbol{y}}) \leftarrow \mathrm{WSO}(\lambda_1, \lambda_2)$
    if $\mathrm{G}_{\mathrm{LB}} < \bar{y}_1 \bar{y}_2$ then
        $\mathrm{G}_{\mathrm{LB}} \leftarrow \bar{y}_1 \bar{y}_2$, $(\boldsymbol{x}^*, \boldsymbol{y}^*) \leftarrow (\bar{\boldsymbol{x}}, \bar{\boldsymbol{y}})$
    $\textsc{Cuts}.add(\lambda_1 y_1 + \lambda_2 y_2 \leq \lambda_1 \bar{y}_1 + \lambda_2 \bar{y}_2)$
    if $\lambda_1 \bar{y}_1 + \lambda_2 \bar{y}_2 > \lambda_1 y_1^1 + \lambda_2 y_2^1$ then
        $Q.add(\boldsymbol{R}(\boldsymbol{y}^1, \bar{\boldsymbol{y}}))$
        $Q.add(\boldsymbol{R}(\bar{\boldsymbol{y}}, \boldsymbol{y}^2))$
return $\boldsymbol{x}^*, \boldsymbol{y}^*, \mathrm{G}_{\mathrm{LB}}, \textsc{Cuts}$
```

R360 with two Intel Xeon E5-2650 2.2 GHz 12-Core Processors (30MB), 128GB RAM, the RedHat Enterprise Linux 6.8 operating system, and using a single thread. Both the relative and optimality gap are set to 10^{-6} in both solution methods, i.e., our branch-and-bound algorithm and the CPLEX-MI-SOCP. Also, a time limit of 7,200 seconds is imposed for each instance in both solution methods. Furthermore, if the preprocessing is active then a time limit of $0.1n$ seconds is imposed for it. Note that the preprocessing can be used both in our proposed branch-and-bound algorithm and CPLEX-MI-SOCP. Consequently, if the preprocessing is active then a time limit of $7200 - t$ seconds is imposed for a solution method where t is the time (in seconds) used by the preprocessing (note that $t \leq 0.1n$). The code and instances used in this study can all be found at https://goo.gl/p7ezR5 and https://goo.gl/bgHciY, respectively. Next, we explain how the instances are generated.

Since pure integer instances can be linearized easily by introducing additional sets of variables and constraints, they can be solved as pure integer linear programs by commercial solvers. Consequently, in this computational study, all instances involve continuous decision variables, i.e., $n_c > 0$ for all instances. In particular, this computational study is conducted over two sets of instances each with 80 randomly generated instances. In both sets, $n_c = 0.5n$ for all instances. However, in the first set, there is no general integer variables, i.e, $n_i = 0$ and $n_b = 0.5n$, and in the second set, there is no binary decision variable, i.e., i.e, $n_b = 0$ and $n_i = 0.5n$. Each set contains 16 subclasses of instances based on the dimensions of the matrix $A_{m\times n}$, and each subclass contains 5 instances. Specially, we assume that $m \in \{200, 400, 800, 1600\}$ and $n = \alpha m$ where $\alpha \in \{0.5, 1, 1.5, 2\}$. For example, the subclass 200×100 implies that $m = 200$ and $n = 100$, i.e., $\alpha = 0.5$. The sparsity of matrix A is set to 50%. The components of vector $\boldsymbol{b}$ and the entries of matrix A are randomly drawn from discrete uniform distributions $[50, 200]$ and $[10, 30]$, respectively. We set the components of vector $\boldsymbol{d}$ to zero. The sparsity of each row of the matrix D was also set to 50% and its components were drawn randomly from a discrete uniform distribution $[1, 10]$. Note that, since all constraints of the set $\mathcal{X}$ are inequality constraints and all coefficients of matrix A are nonnegative, the set $\mathcal{X}$ is bounded. Next, we explain how Problem (2) can be reformulated as mixed integer SOCP.

By introducing a new non-negative variable γ and introducing a *geometric mean* constraint, Problem (2) can be reformulated as follows:

$$\begin{aligned}
&\max \gamma\\
&\text{s.t. } 0 \leq \gamma \leq \Big(\prod_{i=1}^{p} y_i\Big)^{\frac{1}{p}}\\
&\qquad \boldsymbol{y} = D\boldsymbol{x} + \boldsymbol{d}\\
&\qquad A\boldsymbol{x} \leq \boldsymbol{b}\\
&\qquad \boldsymbol{x}, \boldsymbol{y} \geq \boldsymbol{0}, \quad \boldsymbol{x} \in \mathbb{R}^{n_c} \times \mathbb{B}^{n_b} \times \mathbb{Z}^{n_i}, \quad \boldsymbol{y} \in \mathbb{R}^{p}, .
\end{aligned}$$

If $\bar{\gamma}$ is the optimal objective value of the reformulated problem, then $\bar{\gamma}^p$ is the optimal objective value of the original formulation. Ben-Tal and Nemirovski (2001) show how an optimization problem of this form can be written as a mixed integer SOCP. Let k be the smallest integer value such that $2^k \geq p$. By introducing a set of non-negative variables and constraints, the geometric mean constraint can be replaced as follows:

$$\begin{aligned}
&\max \gamma\\
&\text{s.t. } 0 \leq \gamma \leq \Gamma\\
&\qquad 0 \leq \Gamma \leq \sqrt{\tau_1^{k-1}\tau_2^{k-1}}
\end{aligned}$$

$$\begin{aligned}
&0 \le \tau_j^l \le \sqrt{\tau_{2j-1}^{l-1}\tau_{2j}^{l-1}} && \text{for } j = 1,\dots,2^{k-l} \text{ and } l = 1,\dots,k-1\\
&0 \le \tau_j^0 = y_j && \text{for } j = 1,\dots,p\\
&0 \le \tau_j^0 = \Gamma && \text{for } j = p+1,\dots,2^k\\
&\boldsymbol{y} = D\boldsymbol{x} + \boldsymbol{d}\\
&A\boldsymbol{x} \le \boldsymbol{b}\\
&\boldsymbol{x},\boldsymbol{y} \ge \boldsymbol{0}, \quad \boldsymbol{x} \in \mathbb{R}^{n_c} \times \mathbb{B}^{n_b} \times \mathbb{Z}^{n_i}, \quad \boldsymbol{y} \in \mathbb{R}^p.
\end{aligned}$$

The above formulation is a mixed integer SOCP since any constraint of the form $\{u, v, w \ge 0 : u \le \sqrt{vw}\}$ is equivalent to $\{u, v, w \ge 0 : \sqrt{u^2 + (\frac{v-w}{2})^2} \le \frac{v+w}{2}\}$.

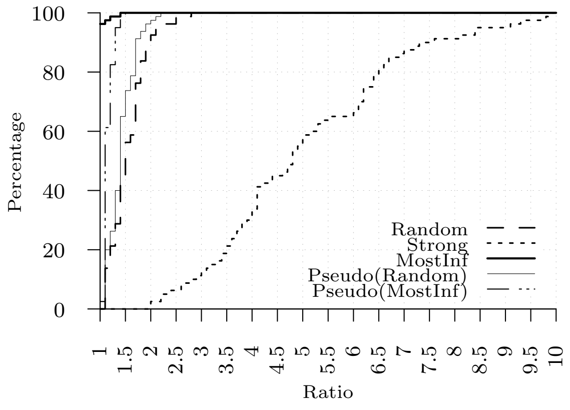
7.1. A performance comparison on instance of set I: mixed binary instances

In this section, we first compare the performance of the proposed branch-and-bound algorithm under different settings obtained by employing the proposed branching strategies, node selecting strategies, and the preprocessing technique on the instances of set I, i.e., those with no general integer variables. After computing a good setting for the proposed algorithm, we then try to find a good setting for the CPLEX-MI-SOCP by exploring whether the proposed preprocessing technique is useful or not. Finally, we compare the performance of both solution methods under their best obtained settings.

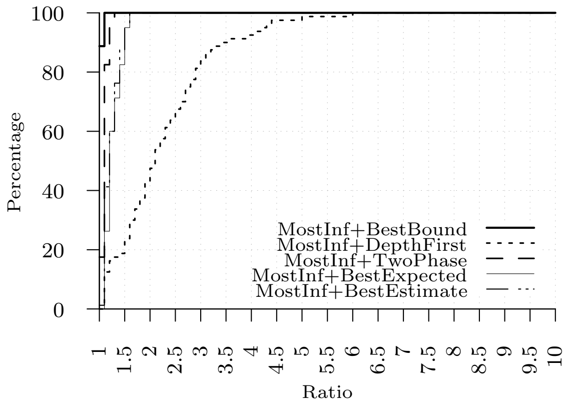
It is worth mentioning that in this computational study, we frequently use *performance profiles* (Dolan and Moré, 2002) to compare different algorithms in term of their solution times. A performance profile presents cumulative distribution functions for a set of algorithms being compared with respect to a specific performance metric. The run time performance profile for a set of algorithms is constructed by computing for each algorithm and for each instance the ratio of the run time of the algorithm on the instance and the minimum of the run times of all algorithms on the instance. The run time performance profile then shows the ratios on the horizontal axis and, on the vertical axis, for each algorithm, the percentage of instances with a ratio that is greater than or equal to the ratio on the horizontal axis. This implies that values in the upper left-hand corner of the graph indicate the best performance.

The run time performance profile of the proposed branch-and-bound algorithm for different branching settings is shown in Figure 4a. It is evident that the most infeasible branching strategy performs the best for the set I of instances, and it outperforms the reliability branching by a factor of up to 10. So, in the remaining, we set the branching strategy to the most infeasible. The run time performance profile of the proposed branch-and-bound algorithm for different node selecting settings is shown in Figure 4b. Observe that the best-bound search strategy performs the best, and it outperforms the depth-first search strategy by a factor of up to 6.2. So, in the remaining, we set the node selecting strategy to the best-bound search strategy.

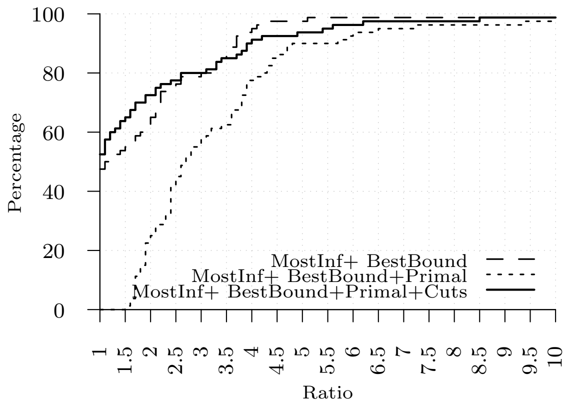
The run time performance profile of the proposed branch-and-bound algorithm for different enhancements including producing an initial primal bound and producing cuts are shown in Figure 4c. Observe that the performance of the algorithm with no enhancement seems to be similar to the performance of the algorithm when both generating cuts and providing an initial primal bound are used. Consequently, there are two best settings for the proposed branch-and-bound algorithm. It is worth noting that the preprocessing operation is a time-consuming operation. So, we observe that if we only use the preprocessing for providing an initial primal bound then the performance of the algorithm decreases.



(a) Branching strategies



(b) Node selecting strategies



(c) Enhancements

Figure 4: The run time performance profile of the proposed branch-and-bound algorithm for different settings on mixed binary instances

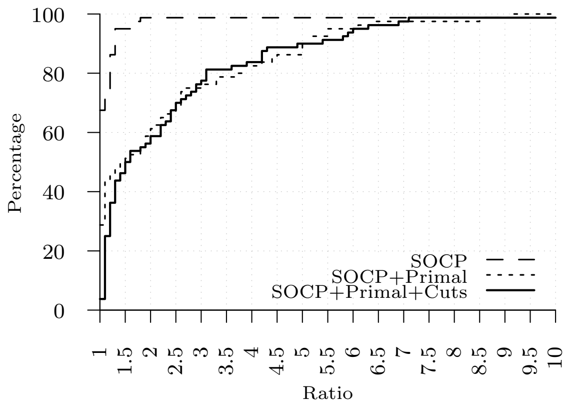


Figure 5: The run time performance profile of CPLEX-MI-SOCP for different settings on mixed binary instances

The run time performance profile of CPLEX-MI-SOCP for different enhancements including producing an initial primal bound and producing cuts are shown in Figure 5. We observe that the performance of CPLEX-MI-SOCP with no enhancement seems to be far better than the case where the cuts and/or an initial primal bound are provided to CPLEX-MI-SOCP. So, the best setting for CPLEX-MI-SOCP is to make the preprocessing inactive.

A detailed comparison between the performance of the best obtained settings for the proposed branch-and-bound algorithm and the CPLEX-MI-SOCP are shown Table 1 where '#N' is the number of nodes, '#LPs' is the number of (single-objective) linear programs solved, 'T(sec.)' shows the solution time in seconds, and finally '%G' shows the optimality gap percentage. Note that in this table averages over 5 instances are reported. Note too that, as discussed earlier, there are two best settings for the proposed branch-and-bound algorithm. So, in Table 1, 'B&B' is used for the case with no preprocessing and 'B&B+ Preprocessing' is used for the case that providing cuts and an initial primal bound is active.

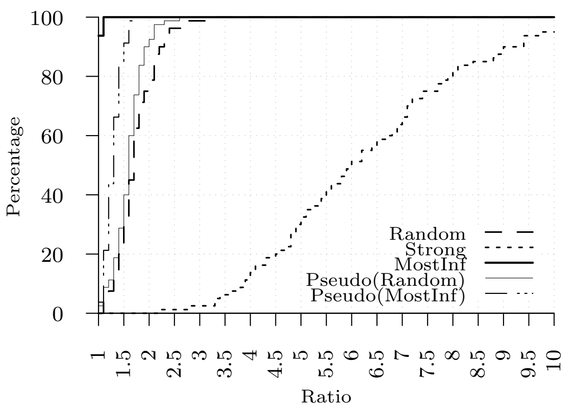
From Table 1, we observe that for small-sized instances, i.e., the classes with 200 and 400 constraints, CPLEX-MI-SOCP seems to perform the best on average. However, as the number of constraints increases, the performance of CPLEX-MI-SOCP decreases dramatically. For classes with 800 and 1,600 constraints, the solution time of the B&B+Preprocessing is better by a factor of around 2 on average. It is worth mentioning that both B&B and CPLEX-MI-SOCP fail to solve instances with 2,400 variables and 3,600 variables in the time limit, i.e., 7,200 seconds. The only algorithm that was able to solve these instances to optimality is B&B+preprocessing.

Table 1: Performance comparison between the best settings of the branch-and-bound algorithm and CPLEX-MI-SOCP on the mixed binary instances

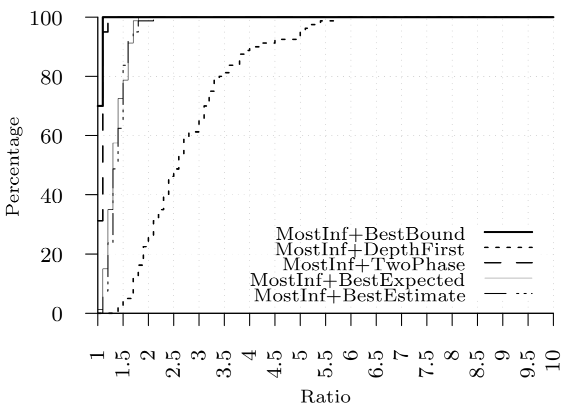
m × n	**B&B**				**B&B+Preprocessing**				**CPLEX-MI- SOCP**		
	#N	**#LPs**	**T(sec.)**	**%G**	**#N**	**LPs**	**T(sec.)**	**%G**	**#N**	**T(sec.)**	**%G**
200 × 100	31.8	1,019.2	2.2	0	5.0	272.2	10.3	0	30.0	2.7	0
200 × 200	28.6	884.4	4.1	0	8.6	370.6	21.0	0	27.0	4.5	0
200 × 300	107.0	3,180.4	24.4	0	40.6	1,460.6	37.3	0	86.2	13.9	0
200 × 400	59.4	1,704.8	20.1	0	52.2	1,637.0	51.1	0	50.6	13.9	0
Avg	**56.7**	**1,697.2**	**12.7**	**0.0**	**26.6**	**935.1**	**29.9**	**0.0**	**48.4**	**8.8**	**0.0**
400 × 200	75.8	2,137.2	20.5	0	67.0	1,861.8	32.5	0	47.4	18.9	0
400 × 400	70.6	1,943.6	45.9	0	73.4	2,455.0	72.0	0	57.8	36.1	0
400 × 600	66.2	1,826.4	85.5	0	50.6	1,597.8	92.0	0	63.8	61.8	0
400 × 800	81.4	2,208.8	174.7	0	70.6	2,151.0	140	0	73.8	93.6	0
Avg	**73.5**	**2,029.0**	**81.6**	**0.0**	**65.4**	**2,016.4**	**84.1**	**0.0**	**60.7**	**52.6**	**0.0**
800 × 400	64.2	1,764.4	90.1	0	36.6	1,133.0	70.0	0	62.6	137.4	0
800 × 800	101.8	2,493.6	457.7	0	105.8	2,993.8	258.0	0	104.4	473.5	0
800 × 1200	81.4	2,153.6	898.5	0	83.4	2,329.8	358.8	0	100.4	792.2	0
800 × 1600	76.2	1,983.6	1,496.9	0	79.4	2,328.2	488.7	0	96.4	1,068.6	0
Avg	**80.9**	**2,098.8**	**735.8**	**0.0**	**76.3**	**2,196.2**	**293.9**	**0.0**	**90.95**	**617.9**	**0.0**
1600 × 800	65.8	1,652.8	698.2	0	67.0	2,060.2	373.6	0	62.2	1,118.1	0
1600 × 1600	64.2	1,602.0	2,875.0	0	56.2	1,503.4	681.9	0	86.6	3,189.2	0
1600 × 2400	64.2	1,758.8	6,793.2	-	85.0	2,571.8	1,851.6	0	113.6	6,903.8	0.04
1600 × 3200	38.2	976.4	7,200.0	-	93.4	2,784.6	2,944.1	0	86.0	7,200.0	0.09
Avg	**58.1**	**1,497.5**	**4,391.7**	**-**	**75.4**	**2,230.0**	**1,462.8**	**0.0**	**87.1**	**4,602.9**	**0.0**

7.2. A performance comparison on instance of set II: mixed general integer instances

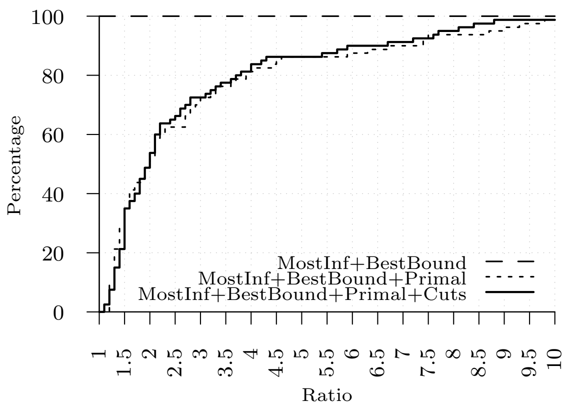
Similar to the previous section, in this section, we first compare the performance of the proposed branch-and-bound algorithm under different settings obtained by employing the proposed branching strategies, node selecting strategies, and the preprocessing technique on the instances of set II, i.e., those with no binary variables. After computing a good setting for the proposed algorithm, we then try to find a good setting for the CPLEX-MI-SOCP by exploring whether the proposed preprocessing technique is useful or not. Finally, we compare the performance of both solution methods under their best obtained settings.



(a) Branching strategies



(b) Node selecting strategies



(c) Enhancements

Figure 6: The run time performance profile of the proposed branch-and-bound algorithm for different settings on mixed integer instances

The run time performance profile of the proposed branch-and-bound algorithm for different branching settings is shown in Figure 6a. Again, it is evident that the most infeasible branching strategy performs the best for the set II of instances. So, in the remaining, we set the branching strategy to the most infeasible. The run time performance profile of the proposed branch-and-bound algorithm for different node selecting settings is shown in Figure 6b. Based on this figure, in the remaining, we set the node selecting strategy to the best-bound search strategy because it performs the best. The run time performance profile of the proposed branch-and-bound algorithm for different enhancements including producing an initial primal bound and cuts are shown in Figure 6c. Observe that the performance of the algorithm with no enhancement performs the best. So, in the remaining, the preprocessing is inactive for the proposed branch-and-bound algorithm.

The run time performance profile of CPLEX-MI-SOCP for different enhancements including producing an initial primal bound and producing cuts are shown in Figure 5. We again observe

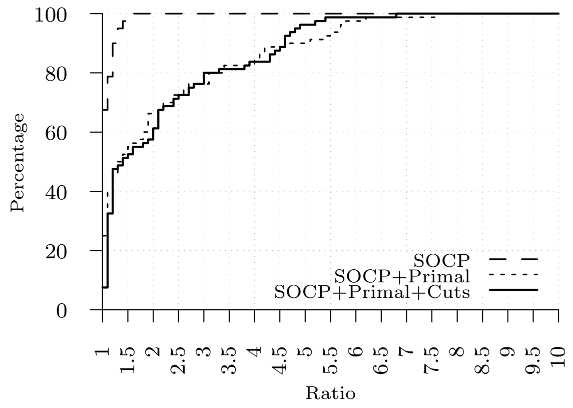


Figure 7: The run time performance profile of CPLEX-MI-SOCP for different settings on mixed integer instances

that the performance of CPLEX-MI-SOCP with no enhancement seems to be far better than the case where the cuts and/or an initial primal bound are provided to CPLEX-MI-SOCP. So, the best setting for CPLEX-MI-SOCP is to make the preprocessing inactive.

A detailed comparison between the performance of the best obtained settings for the proposed branch-and-bound algorithm and the CPLEX-MI-SOCP are shown Table 2. Observe that B&B performs the best across all classes. In fact, B&B outperforms CPLEX-MI-SOCP by a factor of around 2 for all instances on average. Also, we again observe that CPLEX-MI-SOCP fails to solve instances with 2,400 variables and 3,600 variables in the time limit, i.e., 7,200 seconds.

Table 2: Performance comparison between the best settings of the branch-and-bound algorithm and CPLEX-MI-SOCP on the mixed integer instances

m × n	**B&B**				**CPLEX-MI-SCOP**		
	#N	**#LPs**	**T(sec.)**	**%G**	**#N**	**T(sec.)**	**%G**
200 × 100	39.0	1,169.2	2.1	0	28.2	2.3	0
200 × 200	52.2	1,586.8	5.3	0	41.4	6.4	0
200 × 300	38.2	1,165.2	6.0	0	34.0	8.2	0
200 × 400	77.4	2,190.0	14.7	0	68.0	17.0	0
Avg	**51.7**	**1,527.8**	**7.0**	**0.0**	**42.9**	**8.5**	**0.0**
400 × 200	46.6	1,457.2	10.1	0	40.4	17.9	0
400 × 400	74.6	2,101.2	28.7	0	59.8	38.9	0
400 × 600	90.6	2,601.6	54.4	0	85.6	88.5	0
400 × 800	115.4	3,252.0	93.7	0	119.2	157.0	0
Avg	**81.8**	**2,353.0**	**46.7**	**0.0**	**76.2**	**75.6**	**0.0**
800 × 400	96.6	2,592.4	74.4	0	77.6	185.8	0
800 × 800	89.4	2,498.0	154.3	0	110.4	524.0	0
800 × 1200	73.4	2,029.6	206.7	0	94.6	680.5	0
800 × 1600	86.2	2,155.2	315.9	0	95.8	914.3	0
Avg	**86.4**	**2,318.8**	**187.8**	**0.0**	**94.6**	**576.2**	**0.0**
1600 × 800	73.4	1,956.4	285.2	0	79.6	1,410.1	0
1600 × 1600	82.2	2,144.0	772.5	0	94.6	3,493.7	0
1600 × 2400	95.4	2,388.8	1,491.5	0	95.8	6,202.1	0.02
1600 × 3200	104.2	2,484.8	2,223.0	0	75.2	7,200.0	0.11
Avg	**88.8**	**2,243.5**	**1,193.6**	**0.0**	**86.3**	**4,576.7**	**0.0**

8. Final remarks

We developed a multi-objective linear programming based branch-and-bound algorithm for solving a class of mixed integer linear maximum multiplicative programs. This class of optimization problems has only one objective function and it can be solved directly by a commercial mixed integer second-order cone programming solver. However, it was shown that the proposed branch-and-bound algorithm outperforms such a solver by a factor of around 2 on average. Using an extensive computational study, different branching and node selecting strategies as well as enhancement techniques were explored. It was shown that the most infeasible branching and best-bound search strategies perform the best for the proposed branch-and-bound algorithm. However, enhancement techniques were only useful for mixed binary instances. One drawback of the proposed method is that it can only be applied to mixed integer linear maximum multiplicative programs in which its objective function involves a bi-linear term. Therefores, developing a multi-objective optimization based algorithm for cases where the objective function involves a multi-linear term can be a future research direction.

References

Achterberg, T., Koch, T., Martin, A., 2005. Branching rules revisited. Operations Research Letters 33 (1), 42–54.

Aneja, Y. P., Nair, K. P. K., 1979. Bicriteria transportation problem. Management Science 27, 73–78.

Ardakan, M. A., Sima, M., Hamadani, A. Z., Coit, D. W., 2016. A novel strategy for redundant components in reliability–redundancy allocation problems. IIE Transactions 48 (11), 1043–1057.

Belotti, P., Kirches, C., Leyffer, S., Linderoth, J., Luedtke, J., Mahajan, A., 2013. Mixed-integer nonlinear optimization. Acta Numerica 22, 1–131.

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

Bénichou, M., Gauthier, J.-M., Girodet, P., Hentges, G., Ribière, G., Vincent, O., 1971. Experiments in mixed-integer linear programming. Mathematical Programming 1 (1), 76–94.

Benson, H. P., 1984. Optimization over the efficient set. Journal of Mathematical Analysis and Applications 98 (2), 562–580.

Boland, N., Charkhgard, H., Savelsbergh, M., 2017a. A new method for optimizing a linear function over the efficient set of a multiobjective integer program. European Journal of Operational Research 260 (3), 904 – 919.

Boland, N., Charkhgard, H., Savelsbergh, M., 2017b. The quadrant shrinking method: A simple and efficient algorithm for solving tri-objective integer programs. European Journal of Operational Research 260 (3), 873 – 885.

Bonami, P., Lee, J., Leyffer, S., Wächter, A., 2011. More branch-and-bound experiments in convex nonlinear integer programming. Preprint ANL/MCS-P1949-0911, Argonne National Laboratory, Mathematics and Computer Science Division.

Chakrabarty, D., Devanur, N., Vazirani, V. V., 2006. New results on rationality and strongly polynomial time solvability in Eisenberg-Gale markets. In: Internet and Network Economics. Vol. 4286 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 239–250.

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`.

Dächert, K., Klamroth, K., 2014. A linear bound on the number of scalarizations needed to solve discrete tricriteria optimization problems. Journal of Global Optimization, 1–34.

Dolan, E. D., Moré, J. J., 2002. Benchmarking optimization software with performance profiles. Mathematical Programming 91(2), 201–213.

Ehrgott, M., 2005. Multicriteria optimization, 2nd Edition. Springer, New York.

Ehrgott, M., Gandibleux, X., 2007. Bound sets for biobjective combinatorial optimization problems. Computers & Operations Research 34 (9), 2674 – 2694.

Eisenberg, E., Gale, D., 1959. Consensus of subjective probabilities: The pari-mutuel method. The Annals of Mathematical Statistics 30 (1), 165–168.

Eusébio, A., Figueira, J., Ehrgott, M., 2014. On finding representative non-dominated points for bi-objective integer network flow problems. Computers & Operations Research 48, 1 – 10.

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

Gao, Y., Xu, C., Yang, Y., 2006. An outcome-space finite algorithm for solving linear multiplicative programming. Applied Mathematics and Computation 179 (2), 494 – 505.

Goux, J.-P., Leyffer, S., 2002. Solving large minlps on computational grids. Optimization and Engineering 3 (3), 327–346.

Grötschel, M., Lovasz, L., Schrijver, A., 1988. Geometric Algorithms and Combinatorial Optimization. Springer-Verlag, Berlin.

ILOG, I., 2003. Cplex 9.0 reference manual. ILOG CPLEX Division.

Jain, K., Vazirani, V. V., 2007. Eisenberg-Gale markets: Algorithms and structural properties. In: Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing. STOC '07. ACM, New York, NY, USA, pp. 364–373.

Jorge, J. M., 2009. An algorithm for optimizing a linear function over an integer efficient set. European Journal of Operational Research 195 (1), 98–103.

Kirlik, G., Sayın, S., 2014. A new algorithm for generating all nondominated solutions of multiobjective discrete optimization problems. European Journal of Operational Research 232 (3), 479 – 488.

Land, A. H., Doig, A. G., 1960. An automatic method of solving discrete programming problems. Econometrica 28 (3), 497–520.

Little, J. D. C., Murty, K. G., Sweeney, D. W., Karel, C., 1963. An algorithm for the traveling salesman problem. Operations Research 11 (6), 972–989.

Lokman, B., Köksalan, M., 2013. Finding all nondominated points of multi-objective integer programs. Journal of Global Optimization 57 (2), 347–365.

Martin, A., 1999. Integer programs with block structure.

Nash, J. F., 1950. The bargaining problem. Econometrica 18, 155–162.

Nash, J. F., 1953. Two-person cooperative games. Econometrica 21, 128–140.

Özlen, M., Burton, B. A., MacRae, C. A. G., 2013. Multi-objective integer programming: An improved recursive algorithm. Journal of Optimization Theory and Applications10.1007/s10957-013-0364-y.

Özpeynirci, Ö., Köksalan, M., 2010. An exact algorithm for finding extreme supported nondominated points of multiobjective mixed integer programs. Management Science 56 (12), 2302–2315.

Przybylski, A., Gandibleux, X., 2017. Multi-objective branch and bound. European Journal of Operational Research 260 (3), 856 – 872.

Ryoo, H.-S., Sahinidis, N. V., 2003. Global optimization of multiplicative programs. Journal of Global Optimization 26 (4), 387–418.

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.

Soylu, B., Yıldız, G. B., 2016. An exact algorithm for biobjective mixed integer linear programming problems. Computers & Operations Research 72, 204–213.

Vazirani, V. V., 2012a. The notion of a rational convex program, and an algorithm for the Arrow-Debreu Nash bargaining game. J. ACM 59.

Vazirani, V. V., 2012b. Rational convex programs and efficient algorithms for 2-player Nash and nonsymmetric bargaining games. SIAM J. discrete math 26(3), 896–918.

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