MSEA.jl: A Multi-Stage Exact Algorithm for Bi-objective Pure Integer Linear Programming in Julia

Aritra Pal[a], Hadi Charkhgard[b,*]

[a]BNSF Railway Company, Fort Worth, TX, 76131 USA
[b]Department of Industrial and Management Systems Engineering, University of South Florida, Tampa, FL, 33620 USA

Abstract

We present a new exact method for bi-objective pure integer linear programming, the so-called Multi-Stage Exact Algorithm (MSEA). The method combines several existing exact and approximate algorithms in the literature to compute the entire nondominated frontier of any bi-objective pure integer linear program. Each algorithm available in MSEA has multiple versions in the literature. Hence, the main contribution of our research is developing a united framework for all these versions in MSEA. The proposed method is available as an open source Julia package, named MSEA.jl, in GitHub. The package is compatible with the popular JuMP modeling language and supports input in LP and MPS file formats. The package also supports execution on multiple processors. Another desirable feature of the package is that users (if interested) can easily customize the package for their specific problems. In other words, users have full control over all settings of the package and they can even call all different versions of the algorithms available in MSEA.jl in any sequence of interest. So, the package has the potential to be used as a research tool for other researchers in order to develop their own custom-built exact solvers. An extensive computational study demonstrates the efficacy of a few settings of MSEA.jl on some existing standard test instances.

Keywords: bi-objective pure integer linear programming, exact algorithms, heuristic algorithms, parallelization

---

1. Introduction

Bi-Objective Pure Integer Linear Programs (BOPILPs) are optimization problems in which all decision variables are integer and both objective functions and all constraints are linear. The goal in bi-objective pure integer linear programming is to identify the entire or a proportion of the *nondominated frontier*. Each point in the nondominated frontier is the vector of the objective function values of an *efficient* solution, i.e., a feasible solution in which it is impossible to improve the value of one objective without making the value of the other objective worse.

In the last few years, significant advances have been made in the development of more effective algorithms for solving multi-objective integer linear programs, see for instance Boland et al. (2017a,b); Lokman and Köksalan (2013); Özpeynirci and Köksalan (2010); Soylu and Yıldız (2016);

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

Kirlik and Sayın (2014); Dächert et al. (2012); Dchert et al. (2017); Przybylski and Gandibleux (2017); Ehrgott and Gandibleux (2007); Stidsen et al. (2014) and Eusébio et al. (2014). Many of the recently developed algorithms fall into the category of criterion space search algorithms, i.e., those that work in the space of objective functions' values. Such algorithms find a nondominated point by solving a sequence of single-objective optimization problems. After computing a nondominated point, these algorithms remove the dominated parts of the criterion space (based on the obtained nondominated point) and search for not-yet-found nondominated points in the remaining parts.

In light of the above, by studying the literature of criterion space search algorithms for BOPILPs, one can make two important observations. The first one is that, in the relevant literature, the parts of the criterion space that are not dominated by the obtained nondominated points are often decomposed into some rectangular shapes (or simply *boxes*) that can be explored independently. So, the main difference between the recently developed algorithms is often only the search mechanism used for finding a nondominated point in a given box, i.e., the sequence of single-objective optimization problems. The second observation is that for a given problem, combining some of the exact algorithms can result in a faster method to solve the problem (see for instance Leitner et al. (2016); Dai and Charkhgard (2018)).

These two observations suggest that developing a solver that allows users to combine many (if not all) of the criterion space search methods can be valuable. This is mainly because users can then use such a solver as a research tool for finding the best possible way for combining the existing algorithms for their specific problems. In fact this is precisely the main contribution of our research. Specifically, we develop an algorithm the so-called Multi-Stage Exact Algorithm (MSEA) for solving BOPILPs. The algorithm decomposes the criterion spaces into boxes similar to (almost all) criterion spaces search algorithms after finding nondominated point(s), but it allows users to explore boxes by their preferred search mechanisms.

The available search mechanisms in MSEA are all variants of the ones used in existing algorithms including the ϵ-constraint method (Chankong and Haimes, 1983), the balanced box method (Boland et al., 2015), and the perpendicular search method (Chalmet et al., 1986). This implies that MSEA allows users to combine any variants of the ϵ-constraint method, the balanced box method, and the perpendicular search method in any desirable way. The Julia implementation of the algorithm, named MSEA.jl, is available in GitHub, the most popular open-source software repository. MSEA.jl has several other desirable characteristics as well:

- It employs two recently developed generic heuristic approaches, including the feasibility pump based heuristic (Pal and Charkhgard, 2018a,b) and the multi-directional local search (Tricoire, 2012), to compute an approximate nondominated frontier. The information obtained from the approximate frontier will be used by the algorithm when computing the entire nondominated frontier. Users have an option to activate any of the heuristics (if they want). Also, users have an option to provide their own approximate nondominated frontier as an input file.

- The package is compatible with the popular JuMP modeling language (Dunning et al., 2017; Lubin and Dunning, 2015) and supports input in LP and MPS file formats.

- The package supports execution on multiple processors. To employ parallelization, several options are available in which users can choose any of them. It is worth mentioning that several studies have been conducted about parallelization for evolutionary algorithms in multi-objective optimization (see for instance Yu et al. (2017) and Pedroso et al. (2017)). In fact the feasibility pump based heuristic also employs parallelization. However, unfortunately,

this topic has been almost untouched in the literature of exact algorithms. The recent study conducted by Pettersson and Özlen (2017) is one of the few papers (if not the only one) in this scope.

The rest of this paper is organized as follows. In Section 2, a high-level description of the algorithm and some important concepts and notations are explained. The algorithm contains three main phases. A detailed description of each of these phases are explained in Sections 3-5, respectively. In Section 6, different strategies for alternating between search mechanisms are introduced. In Section 7, an extensive computational study has been conducted. Finally, in Section 8, some concluding remarks are given.

2. The framework

A bi-objective pure integer linear program can be stated as follows:

$$\min_{\boldsymbol{x}\in\mathcal{X}} \{z_1(\boldsymbol{x}), z_2(\boldsymbol{x})\},$$

where $\mathcal{X} := \{\boldsymbol{x} \in \mathbb{Z}^n_+ : A\boldsymbol{x} \leq \boldsymbol{b}\}$ represents the *feasible set in the decision space*, and $z_i(\boldsymbol{x}) := \boldsymbol{c}^i\boldsymbol{x}$ for each $i = 1, 2$ represents a linear objective function. The image $\mathcal{Y}$ of $\mathcal{X}$ under vector-valued function $\boldsymbol{z} = \{z_1, z_2\}$ represents the *feasible set in the objective/criterion space*, i.e., $\mathcal{Y} := \boldsymbol{z}(\mathcal{X}) := \{\boldsymbol{y} \in \mathbb{R}^p : \boldsymbol{y} = \boldsymbol{z}(\boldsymbol{x}) \text{ for some } \boldsymbol{x} \in \mathcal{X}\}$. We denote the linear programming (LP) relaxation of $\mathcal{X}$ by $LP(\mathcal{X})$, and we assume that it is *bounded*. We also assume that all coefficients/parameters are integers, i.e., $A \in \mathbb{Z}^{m\times n}$, $\boldsymbol{b} \in \mathbb{Z}^m$, and $\boldsymbol{c}^i \in \mathbb{Z}^n$ for each $i = 1, 2$.

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 $z_1(\boldsymbol{x}) \leq z_1(\boldsymbol{x}')$ and $z_2(\boldsymbol{x}) < z_2(\boldsymbol{x}')$ or $z_1(\boldsymbol{x}) < z_1(\boldsymbol{x}')$ and $z_2(\boldsymbol{x}) \leq z_2(\boldsymbol{x}')$. If $\boldsymbol{x}'$ is efficient, then $\boldsymbol{z}(\boldsymbol{x}')$ is called a nondominated point. The set of all efficient solutions $\boldsymbol{x}' \in \mathcal{X}$ is denoted by $\mathcal{X}_E$. The set of all nondominated points $\boldsymbol{z}(\boldsymbol{x}') \in \mathcal{X}$ for some $\boldsymbol{x}' \in \mathcal{X}_E$ is denoted by $\mathcal{Y}_N$ and referred to as the nondominated frontier.*

Overall, bi-objective pure integer linear programming is concerned with finding *all* nondominated points. Specifically, $\min_{\boldsymbol{x}\in\mathcal{X}} \boldsymbol{z}(\boldsymbol{x})$ is defined to be precisely $\mathcal{Y}_N$. The set of all nondominated points of a BOPILP is finite (since by assumption $LP(\mathcal{X})$ is bounded). Next, we explain the framework of MSEA.jl in detail.

MSEA.jl takes several optional parameters as inputs that we will discuss about them whenever appropriate in this paper. The first parameter is the number of threads, denoted by $NumThreads$. To the best of our knowledge, investigating the effect of parallelization is untouched in the literature of solution approaches for multi-objective optimization (Pal and Charkhgard, 2018a,b). So, this study is one of the first studies in this scope.

MSEA.jl maintains a list of nondominated points found during the course of the algorithm which is denoted by L. If the algorithm terminates naturally (i.e., does not terminate because the time limit is reached), L will contain all nondominated points of the problem at the time of termination. Overall, MSEA.jl has three phases: including Approximation, Initialization, and Parallelization. We will discuss about each of these phases in Sections 3-5 in detail. However, in the remaining of this section, we will provide an overview of each phase.

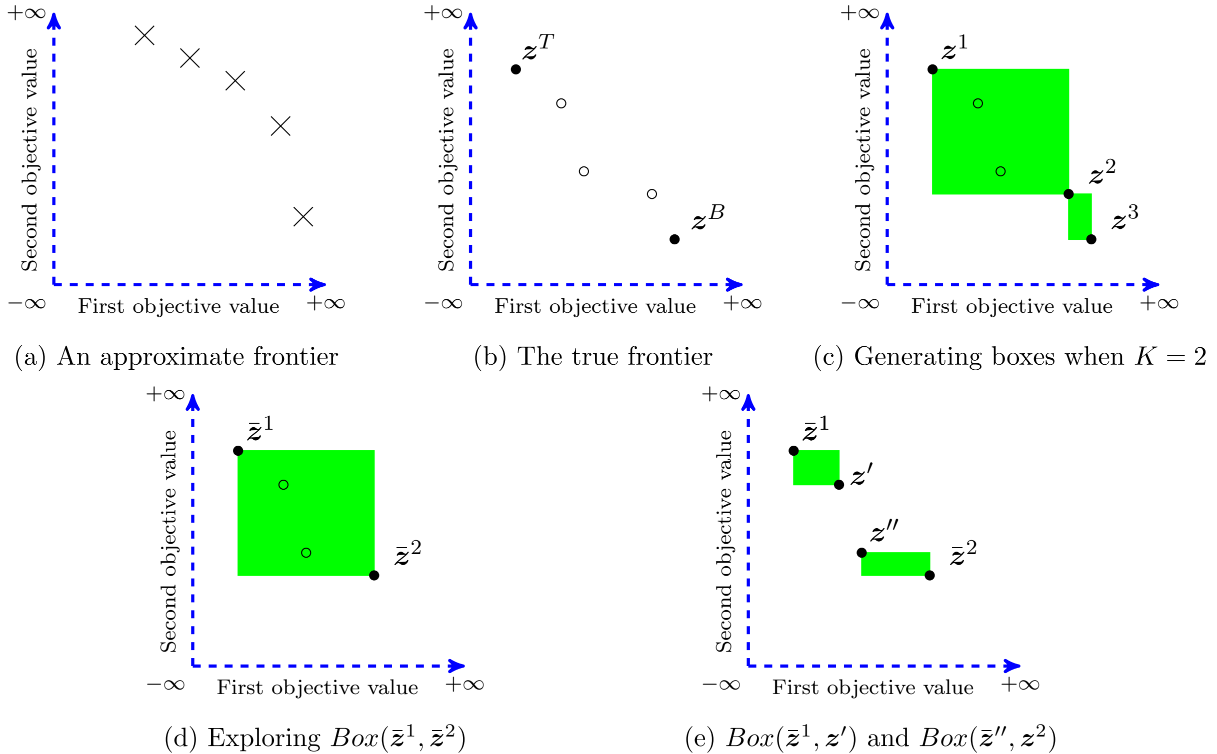
In Phase I (approximation), a set of approximate efficient solutions will be generated which is denoted by $\tilde{\mathcal{X}}_E$. These solutions will be used in other phases for boosting the performance of

Algorithm 1: MSEA($NumThreads, OtherParameters$)

1 $List.create(L)$
2 #**Phase I (Approximation):** Compute an approximate frontier
3 $\tilde{\mathcal{X}}_E \leftarrow$ COMPUTE-APPROXIMATE-EFFICIENT-SOLUTIONS
4 #**Phase II (Initialization):** Generate initial boxes
5 $(\boldsymbol{z}^T, \boldsymbol{z}^B) \leftarrow$ GENERATE-END-POINTS
6 $\big(Box(\boldsymbol{z}^1, \boldsymbol{z}^2), Box(\boldsymbol{z}^2, \boldsymbol{z}^3), \ldots, Box(\boldsymbol{z}^K, \boldsymbol{z}^{K+1})\big) \leftarrow$ GENERATE-BOXES($NumThreads$)
7 $List.add(L, \boldsymbol{z}^1)$; $List.add(L, \boldsymbol{z}^2)$; $\ldots$; $List.add(L, \boldsymbol{z}^K)$; $List.add(L, \boldsymbol{z}^{K+1})$;
8 #**Phase III (Parallelization):** Do using **processor** $k \in \{1, \ldots, K\}$
9 $Queue.create(Q_k)$
10 $Queue.add(Q_k, Box(\boldsymbol{z}^k, \boldsymbol{z}^{k+1}))$
11 **while** *not* $Queue.empty(Q_k)$ **do**
12     $Queue.pop(Q_k, Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2))$
13     $(\boldsymbol{z}', \boldsymbol{z}'') \leftarrow$ SEARCH($Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$)
14   **if** $\boldsymbol{z}' \neq null$ **then**
15         $List.add(L, \boldsymbol{z}')$
16   **if** $Box(\boldsymbol{z}^1, \boldsymbol{z}')$ *can still contain nondominated points* **then**
17             $Queue.add(Q_k, Box(\bar{\boldsymbol{z}}^1, \boldsymbol{z}'))$
18   **if** $\boldsymbol{z}'' \neq null$ **then**
19         $List.add(L, \boldsymbol{z}'')$
20   **if** $Box(\boldsymbol{z}'', \bar{\boldsymbol{z}}^2)$ *can still contain nondominated points* **then**
21             $Queue.add(Q_k, Box(\boldsymbol{z}'', \boldsymbol{z}^2))$
22   **if** $\boldsymbol{z}' \neq null$ *&* $\boldsymbol{z}'' \neq null$ **then**
23   **if** $Box(\boldsymbol{z}', \boldsymbol{z}'')$ *can still contain nondominated points* **then**
24             $Queue.add(Q_k, Box(\boldsymbol{z}', \boldsymbol{z}''))$
25 **return** L

MSEA.jl. In order to create the approximate frontier, the algorithm calls an operation denoted by Compute-Approximate-Efficient-Solutions that will return N approximate efficient solutions (where $N \geq 0$). These solutions are not necessarily the true efficient solutions of the problem but they cannot dominate each other, i.e., for each $v, w \in \{1, \ldots, N\}$ where $v \neq w$ we have that $z_i(\boldsymbol{x}^v) < z_i(\boldsymbol{x}^w)$ and $z_j(\boldsymbol{x}^v) > z_j(\boldsymbol{x}^w)$ where $i, j \in \{1, 2\}$ (see Figure 1a).

In Phase II (initialization), the algorithm first computes the true endpoints of the nondominated frontier, denoted by $\boldsymbol{z}^T$ and $\boldsymbol{z}^B$ (see Figure 1b). After finding the endpoints, the algorithm attempts to find $NumThread - 1$ more nondominated points. However, the algorithm may fail to do so because of its structure (we will explain about it further in Section 4). Consequently, after this attempt, we assume that the algorithm has computed in total $K+1$ nondominated points including $\boldsymbol{z}^T$ and $\boldsymbol{z}^B$ (where $K \leq NumThread$). We denote these points by $\boldsymbol{z}^1, \ldots, \boldsymbol{z}^K, \boldsymbol{z}^{K+1}$. Without loss of generality, we assume that $\boldsymbol{z}^1 = \boldsymbol{z}^T$, $\boldsymbol{z}^{K+1} = \boldsymbol{z}^B$, and for each $k = 1, \ldots, K$ we have that $z_1^k < z_1^{k+1}$ and $z_2^k > z_2^{k+1}$. Because all these points are nondominated and they include the endpoints of the nondominated frontier, it is easy to see that all not-yet-found nondominated points must be in boxes defined by all consecutive points, i.e., $Box(\boldsymbol{z}^1, \boldsymbol{z}^2), Box(\boldsymbol{z}^2, \boldsymbol{z}^3), \ldots, Box(\boldsymbol{z}^K, \boldsymbol{z}^{K+1})$. An illustration of these boxes when $K = 2$ can be found in Figure 1c. The operation required to generate these boxes is denoted by Generate-Boxes($NumThreads$) in MSEA.jl. After creating these boxes, the algorithm adds the obtained nondominated points, i.e., $\boldsymbol{z}^1, \ldots, \boldsymbol{z}^K, \boldsymbol{z}^{K+1}$, to L.



(a) An approximate frontier (b) The true frontier (c) Generating boxes when $K = 2$

(d) Exploring $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$ (e) $Box(\bar{\boldsymbol{z}}^1, \boldsymbol{z}')$ and $Box(\bar{\boldsymbol{z}}'', \boldsymbol{z}^2)$

Figure 1: An illustration of the key components of MSEA.jl

In Phase III (parallelization), the algorithm explores all boxes defined in Phase II in parallel on different processors. Specifically, on processor $k \in \{1, \ldots, K\}$, the algorithm explores $Box(\boldsymbol{z}^k, \boldsymbol{z}^{k+1})$

to find all not-yet-found nondominated points in that box. In order to do so, the algorithm first creates a queue of boxes, denoted by Q_k that will be generated during the course of the algorithm as a result of exploring $Box(\boldsymbol{z}^k, \boldsymbol{z}^{k+1})$. The queue is empty at the beginning and so it will be initialized by adding $Box(\boldsymbol{z}^k, \boldsymbol{z}^{k+1})$. The algorithm next repeats the following steps until Q_k becomes empty, i.e., no more nondominated points can be found.

In each iteration, the algorithm pops out a box from Q_k and denote it by $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$. An illustration of $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$ can be found in Figure 1d. Note that when a box is popped out then that box does not exist in the queue anymore. The algorithm then searches that box to find at most two nondominated points denoted by $\boldsymbol{z}'$ and $\boldsymbol{z}''$ where $\bar{z}^1_1 < z'_1 \leq z''_1 < \bar{z}^2_1$ and $\bar{z}^1_2 > z'_2 \geq z''_2 > \bar{z}^2_2$ (if $\boldsymbol{z}'$ and $\boldsymbol{z}''$ exist).

In light of the above, if $\boldsymbol{z}'$ exists, i.e., it is not null, then the algorithm will add $\boldsymbol{z}'$ to L. Also, it will add a new box denoted by $Box(\boldsymbol{z}^1, \boldsymbol{z}')$ to Q_k if the new box can still contain not-yet-found nondominated points. Similarly, if $\boldsymbol{z}''$ exists, i.e., it is not null, then the algorithm will add $\boldsymbol{z}''$ to L. Also, it will add a new box denoted by $Box(\boldsymbol{z}'', \boldsymbol{z}^2)$ to Q_k if the new box can still contain not-yet-found nondominated points. An illustration of $Box(\boldsymbol{z}^1, \boldsymbol{z}')$ and $Box(\boldsymbol{z}'', \boldsymbol{z}^2)$ can be found in Figure 1e. If $\boldsymbol{z}'$ and $\boldsymbol{z}''$ exist then the algorithm also adds $Box(\boldsymbol{z}', \boldsymbol{z}'')$ to Q_k if it can still contain not-yet-found nondominated points.

A detailed description of the framework of MSEA.jl can be found in Algorithm 1. We conclude this section by making two comments:

- There are a few special cases in which they guarantee that a given box cannot contain a not-yet found nondominated point. The most obvious one is that the length or the width of the box is less than or equal to 1. This is because we have assumed that all the coefficients and variables of a BOPILP are integers. So, this observation can be used in Phase III when adding $Box(\boldsymbol{z}^1, \boldsymbol{z}')$ or $Box(\boldsymbol{z}'', \boldsymbol{z}^2)$ or $Box(\boldsymbol{z}', \boldsymbol{z}'')$ to Q_k. We will discuss about other special cases whenever appropriate in this paper.

- In the developed Julia package, i.e., MSEA.jl, the users are allowed to impose time limits for almost all components of the algorithm, e.g., each of the three phases. Obviously, in that case, the algorithm may not be able to find all nondominated points. Hence, in that case, at the time of termination the algorithm reports an approximation of the true nondominated frontier.

3. Phase I

The MSEA.jl package generates approximate efficient solutions that will be mainly used for warm-starting purposes. Three different ways of generating approximate efficient solutions are embedded in MSEA.jl:

- **User-defined solutions:** Users are allowed to provide an arbitrary number of feasible solutions to MSEA.jl.

- **FPBH.jl**: This is the first Feasibility Pump Based Heuristic (FPBH) for generating approximate efficient solutions for any multi-objective mixed integer linear program with an arbitrary number of objective functions. FPBH is implemented in Julia (`https://goo.gl/SVYNWK`), and in addition to the feasibility pump method, it exploits the underlying ideas of several other algorithms in the literature of both single-objective and multi-objective optimization

including (but not limited to) a weighted sum method, a local search approach, and a local branching method. FPBH.jl also supports execution on multiple processors. The default time limit for FPBH.jl in MSEA.jl is 60 seconds. However, users can change this if they want.

- **MDLS**: The Multi-Directional Local Search (MDLS) is a heuristic approach for generating approximate efficient solutions for any multi-objective pure integer linear program (Tricoire, 2012). Unlike FPBH, the available C++ implementation (`https://goo.gl/BzJ32d`) of MDLS is problem-dependent in a sense that some of its source/header files should be specifically defined for any problem that a user attempts to solve. Fortunately, for instances of the knapsack problem, the corresponding header/source files exist and so they are included in MSEA.jl. So, users can call MDLS when solving Knapsack instances by MSEA.jl. There is no default time limit for MDLS in MSEA.jl because it often terminates in less than 10 seconds.

Overall, users can activate one or several of the above ways to generate approximate efficient solutions in MSEA.jl. At the end of Phase I, the package will combine the results of all the activated ways and remove the dominated or equivalent solutions from them to generate the ultimated set of approximate efficient solutions, i.e., $\tilde{\mathcal{X}}_E$. In other words, if there are two feasible solutions $\boldsymbol{x}'$ and $\boldsymbol{x}''$ with $z_1(\boldsymbol{x}') \leq z_1(\boldsymbol{x}'')$ and $z_1(\boldsymbol{x}') \leq z_2(\boldsymbol{x}'')$ only $\boldsymbol{x}'$ can be in $\tilde{\mathcal{X}}_E$. We again note that $\tilde{\mathcal{X}}_E$ is not necessarily a subset of the true efficient solutions of the problem, i.e., $\mathcal{X}_E$. However, we have that $\tilde{\mathcal{X}}_E \subseteq \mathcal{X}$.

4. Phase II

As mentioned earlier in Section 2, In Phase II, two major operations should be conducted. The first computes the endpoints of the nondominated frontier, i.e., $\boldsymbol{z}^T$ and $\boldsymbol{z}^B$, and the second operation decomposes the criterion space into multiple independent boxes that will be explored in parallel in Phase III. We explain both of these operations in details in this section.

The top endpoint $\boldsymbol{z}^T$ can be computed by solving two (single-objective) Pure Integer Linear Programs (PILPs) in sequence, as follows:

$$z_1^T = \min_{\boldsymbol{x} \in \mathcal{X}} z_1(\boldsymbol{x})$$

(if feasible) followed by

$$z_2^T = \min_{\boldsymbol{x} \in \mathcal{X}} \{z_2(\boldsymbol{x}) : z_1(\boldsymbol{x}) \leq z_1^T\}.$$

As this is an operation that will be called frequently in this paper, we introduce the following notation to represent the process:

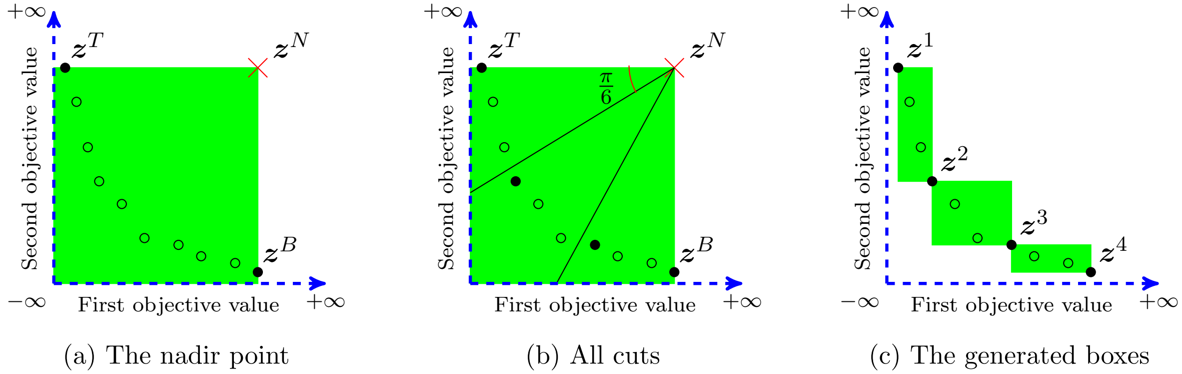
$$\boldsymbol{z}^T = \operatorname{lex}\min_{\boldsymbol{x} \in \mathcal{X}} \{z_1(\boldsymbol{x}), z_2(\boldsymbol{x})\}.$$

As an aside, the solution with minimum value for the first objective function in $\tilde{\mathcal{X}}_E$ (which is obtained in Phase I) will be used (if $\tilde{\mathcal{X}}_E \neq \emptyset$) to warm start the first PILP in the above lexicographic operation. For the second PILP the obtained optimal solution of the first PILP will be used for warm starting.

Similarly, the bottom endpoint $\boldsymbol{z}^B$ can be computed by using:

$$\boldsymbol{z}^B = \operatorname{lex}\min_{\boldsymbol{x} \in \mathcal{X}} \{z_2(\boldsymbol{x}), z_1(\boldsymbol{x})\}.$$

Note that since $\boldsymbol{z}^B$ will be computed after $\boldsymbol{z}^T$, in MSEA.jl, the obtained efficient solution corresponding to $\boldsymbol{z}^T$ will be considered as a warm-start solution for computing $\boldsymbol{z}^B$ (if it has a better quality than those in $\tilde{\mathcal{X}}_E$ in terms of the second objective value).



(a) The nadir point (b) All cuts (c) The generated boxes

Figure 2: An illustration of the proposed decomposition technique when 3 processors are available

In the second operation of Phase II, the criterion space will be decomposed. It is worth mentioning that the default decomposition technique in MSEA.jl is one of the contributions of our research. Let $\boldsymbol{z}^N$ such that $z_1^N = z_1^B$ and $z_2^N = z_2^T$ (which is known as the *nadir point*). An illustration of the nadir point can be found in Figure 2a. For notational convenience, let $M := NumThreads$. In the proposed decomposition technique, the algorithm attempts to divide the criterion space into M segments by adding $M-1$ lines in which their slopes are from the set $\alpha \in \{\frac{\pi}{2M}, \dots, \frac{(M-1)\pi}{2M}\}$ and originates from $\boldsymbol{z}^N$. An illustration of these lines, i.e., $z_2(\boldsymbol{x}) - z_2^N = \tan(\alpha)(z_1(\boldsymbol{x}) - z_1^N)$ for all $\alpha \in \{\frac{\pi}{2M}, \dots, \frac{(M-1)\pi}{2M}\}$, can be found in Figure 2b when $M = 3$. Specifically, for each $v \in \{1, \dots, M-1\}$, the algorithm solves the following two PILPs in sequence, as follows:

$$\tilde{\boldsymbol{x}}^v \in \arg\min_{\boldsymbol{x}\in\mathcal{X}} \{z_1(\boldsymbol{x}) : z_2(\boldsymbol{x}) - z_2^N \leq \tan(\frac{v\pi}{2M})(z_1(\boldsymbol{x}) - z_1^N)\},$$

followed by

$$\boldsymbol{x}^v \in \arg\min_{\boldsymbol{x}\in\mathcal{X}} \{z_1(\boldsymbol{x}) + z_2(\boldsymbol{x}) : z_1(\boldsymbol{x}) \leq z_1(\boldsymbol{x}^v) \text{ and } z_2(\boldsymbol{x}) \leq z_2(\boldsymbol{x}^v)\}.$$

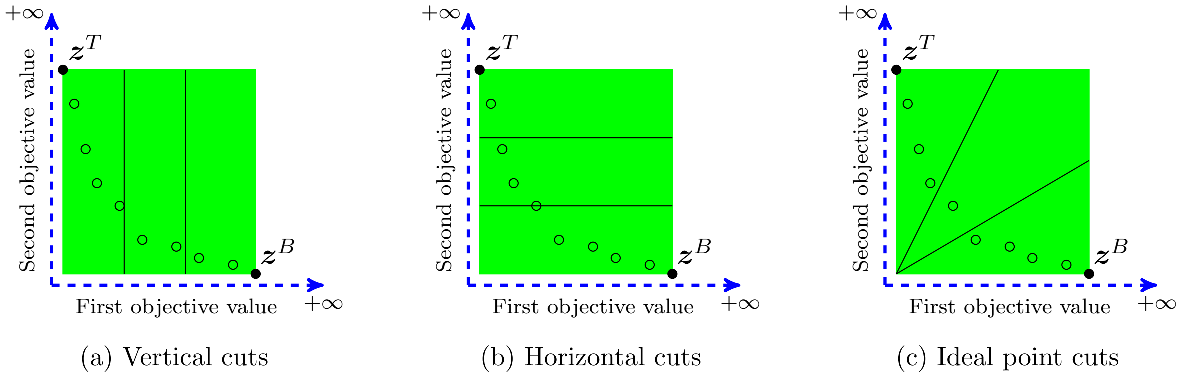
The first PILP attempts to find a feasible solution, denoted by $\tilde{\boldsymbol{x}}^v$, that minimizes the first objective function and its image in the criterion space lies below the line $z_2(\boldsymbol{x}) - z_2^N \leq \tan(\frac{v\pi}{2M})(z_1(\boldsymbol{x}) - z_1^N)$. Obviously, solution $\tilde{\boldsymbol{x}}^v$ is not necessarily an efficient solution. To generate an efficient solution based on $\boldsymbol{x}^v$, the second PILP should be solved. The new PILP attempts to find a feasible solution, denoted by $\boldsymbol{x}^v$, such that $z_1(\boldsymbol{x}^v) \leq z_1(\tilde{\boldsymbol{x}}^v)$ and $z_2(\boldsymbol{x}^v) \leq z_2(\tilde{\boldsymbol{x}}^v)$ and minimizes the summation of both objective functions. The illustration of the obtained efficient solutions, i.e., $\boldsymbol{x}^v$ for all $v \in \{1, \dots, M-1\}$, using the proposed method when $M = 3$ in the criterion space can be found in Figure 2b, i.e., the two filled circles inside the green area. Also, Figure 2c shows the boxes that will be generated at the end of Phase II.

We conclude this section by making a few final comments:

- It is possible that there exist $v, w \in \{1, \dots, M-1\}$ with $v \neq w$ such that $z_1(\boldsymbol{x}^v) = z_1(\boldsymbol{x}^w)$ and $z_2(\boldsymbol{x}^v) = z_2(\boldsymbol{x}^w)$. Obviously in that case, at the end of Phased II, the algorithm will not

be able to generate exactly M independent boxes, i.e., $K < M$ in Algorithm 1. So, in that case, the user has provided M processors but the algorithm is only able to employ K of them.

- The proposed decomposition technique creates at most M independent boxes at the end of Phase II that will be explored in parallel in Phase III. Intuitively, if the created boxes have almost the same number of nondominated points then we can expect that the parallelization to be more effective, i.e., reduce the solution time more. In light of this observation, our numerical results show that the proposed decomposition technique work well in practice (see Section 7). This is mainly because we have observed that, in practice, the nondominated frontier of BOPILP often looks like a sharp convex curve (if we draw an imaginary line between any consecutive nondominated points). So, using the proposed decomposition technique, it is likely that the generated boxes have the same number of nondominated points (see Figure 2c).



(a) Vertical cuts (b) Horizontal cuts (c) Ideal point cuts

Figure 3: An illustration of the other decomposition techniques when 3 processors are available

- MSEA.jl employs other decomposition techniques as well that users can activate them for their specific problems to see which one works the best. Some of the available decomposition techniques are vertical decomposition, horizontal decomposition, and decomposition based on $\boldsymbol{z}^I$ with $z_1^I = z_1^T$ and $z_2^I = z_2^B$ (which is known as the *ideal point*). An illustration of these decomposition techniques can be found in Figure 3.

- MSEA.jl employs CPLEX as a commercial single-objective optimization solver when solving a PILP. So, for Phase II, users of MSEA.jl have the option to turn off all the internal primal heuristics of CPLEX by disabling the following parameters: CPX_PARAM_HEURFREQ and CPX_PARAM_RINSHEUR and CPX_PARAM_FPHEUR. This option can be particularly useful, i.e., can improve the solution time of the algorithm, if the algorithm has found good approximate efficient solutions in Phase I since they can be used for warm-starting purposes.

5. Phase III

There are two main operations in Phase III that requires further explanations. The first one is about the order in which the boxes should be popped out from the queue Q_k (where $k \in \{1, \ldots, K\}$) in each iteration. The second operation is about the search procedure used to explore $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$ in each iteration.

MSEA.jl employs two mechanisms for ordering the boxes in the queue. The default one computes the *minimum dimension* of each box, i.e., the minimum of the length and width of a box, and hence maintains the boxes in non-decreasing order of their minimum dimensions in the queue. The second mechanism maintains the boxes in non-decreasing order of their areas in the queue. Users can activate any of these mechanisms if they are interested. However, no matter which mechanism is used, the algorithm always pops out the first box of the queue in each iteration of Phase III. Overall, both of these mechanisms are helpful when generating all nondominated points is not possible because of imposing time limit on Phase III by users. Obviously, in such a case, the algorithm will report an approximation of the nondominated frontier at the time of termination. So, intuitively, exploring the rectangles in the proposed orders can be helpful in creating high-quality approximations.

To explore $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$ in each iteration, the operation SEARCH($Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$) should be called. MSEA.jl employs three main mechanisms to explore $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$ which will be explained in details next.

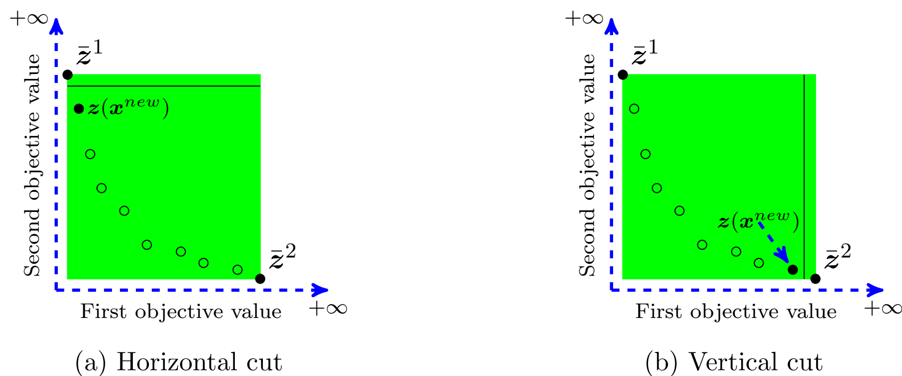
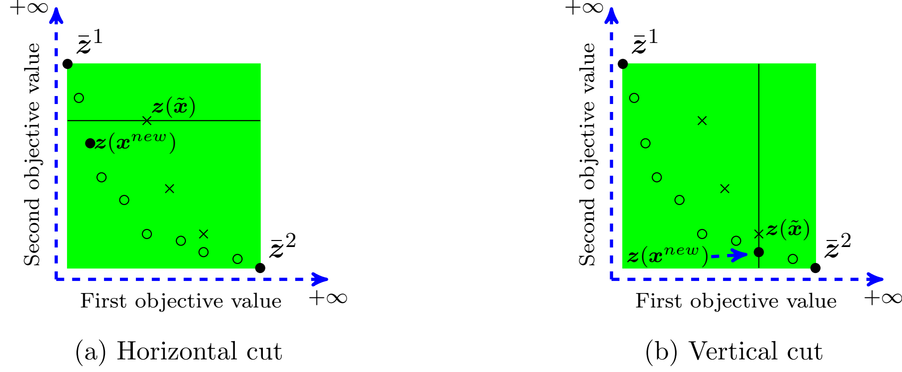


Figure 4: An illustration of the impact of 'Direction' on the Epsilon Search

5.1. Epsilon search

One of the search mechanisms available in MSEA.jl is motivated by the ϵ-constraint method (Chankong and Haimes, 1983). Different variants of this mechanism is frequently used in the literature of the ϵ-constraint method. MSEA.jl employs most (if not all) of the existing variants as well as some new ones and users have the flexibility to switch between them (at any time) if they want (details are explained in Section 6). We define a general operation named as Epsilon Search (ES) that captures all different variants. This operation is denoted by ES($Direction, Type$) which takes two inputs.

'Direction' is either horizontal or vertical. If it is horizontal then the cut line is horizontal and the operation searches for a nondominated point on or below the cut line that has the minimum value for the first objective function (as shown in Figure 4a). If it is vertical then the cut line is vertical and the operation searches for a nondominated point on or left side of the cut line that has the minimum value for the second objective function (as shown in Figure 4b). Note that for the horizontal cut a solution corresponding to $\bar{\boldsymbol{z}}^2$ is feasible so it will used to warm start the search procedure for finding a nondominated point. Similarly, for the vertical cut a solution corresponding to $\bar{\boldsymbol{z}}^1$ is feasible so it will used to warm start the search procedure for finding a nondominated point.



(a) Horizontal cut

(b) Vertical cut

Figure 5: An illustration of the Epsilon Search when 'InitialSolution' is one and a feasible solution $\tilde{\boldsymbol{x}} \in \tilde{\mathcal{X}}_E$ is known

The position of a cut will be determined by the algorithm automatically. In order to do so, the algorithm first searches $\tilde{\mathcal{X}}_E$ for finding a feasible solution that its image in the criterion space lies in the interior of $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$. If such a solution does not exist then the cut will be placed exactly after an endpoint. This implies that for the horizontal direction, the cut $z_2(\boldsymbol{x}) \leq \bar{z}_2^1 - \epsilon$ will be added (see Figure 4a). Also, for the vertical direction, the cut $z_1(\boldsymbol{x}) \leq \bar{z}_1^2 - \epsilon$ will be added (see Figure 4b). Note that because by assumptions, all parameters and variables are integers, without loss of generality, we can set $\epsilon = 1$. However, if there are some feasible solutions then the algorithm will place the cut on the image of one of those solutions in the criterion space. This is motivated by this observation that by doing so, such a solution is expected to be a high-quality feasible solution and can be used to warm start the search for a nondominated point. In the default setting, for the horizontal cut, the algorithm first finds a solution $\tilde{\boldsymbol{x}} \in \tilde{\mathcal{X}}_E$ that has the minimum value for the first objective function among those that their images in the criterion space lie in the interior of $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$. The algorithm then adds the cut $z_2(\boldsymbol{x}) \leq z_2(\tilde{\boldsymbol{x}})$, as shown in Figure 5a. Similarly, for the vertical cut, the algorithm first finds a solution $\tilde{\boldsymbol{x}} \in \tilde{\mathcal{X}}_E$ that has the minimum value for the second objective function among those that their images in the criterion space lie in the interior of $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$. The algorithm then adds the cut $z_1(\boldsymbol{x}) \leq z_1(\tilde{\boldsymbol{x}})$, as shown in Figure 5b.

'Type' shows the type of optimization problems required for finding a nondominated point. ES employs possibles ways for computing a nondominated point:

- *Lexicographic Method 1 (LM1)*: For the horizontal cut, the algorithm solves

$$\operatorname{lex}\min_{x \in \mathcal{X}} \{z_1(\boldsymbol{x}), z_2(\boldsymbol{x}) : \text{cut}\}.$$

  For the vertical cut, the algorithm solves

$$\operatorname{lex}\min_{x \in \mathcal{X}} \{z_2(\boldsymbol{x}), z_1(\boldsymbol{x}) : \text{cut}\}.$$

- *Lexicographic Method 2 (LM2)*: This is similar to LM1 but the only difference is that 'CPLEX MIP emphasis' parameter will be set to 'optimality' for solving the second optimization problem in the "lex min" operation. This is mainly because the optimal solution obtained by solving the first optimization problem is likely to be also optimal for the second optimization problem. Consequently, this small change can possibly help CPLEX to solve the second optimization problem faster.

- *Augmented Method (AM)*: This operation returns exactly the same nondominated point that LM1 and LM2 produce for a given cut. The only difference is that instead of solving two optimization problems, only one optimization problem should be solved (Özlen and Azizoğlu, 2009). For the horizontal cut, the algorithm solves

$$\boldsymbol{x}^{new} \in \arg\min_{x \in \mathcal{X}} \{z_1(\boldsymbol{x}) + \alpha_2 z_2(\boldsymbol{x}) : \text{cut}\},$$

  where $\alpha_2 = 1/(z_2^T - z_2^B + 1)$. The outcome of this operation $\boldsymbol{z}^{new} := \boldsymbol{z}(\boldsymbol{x}^{new})$ is a nondominated point. Similarly, for the horizontal cut, the algorithm solves

$$\boldsymbol{x}^{new} \in \arg\min_{x \in \mathcal{X}} \{\alpha_1 z_1(\boldsymbol{x}) + z_2(\boldsymbol{x}) : \text{cut}\},$$

  where $\alpha_1 = 1/(z_1^B - z_1^T + 1)$. The outcome of this operation $\boldsymbol{z}^{new} := \boldsymbol{z}(\boldsymbol{x}^{new})$ is a nondominated point. As an side, we note that AM sometimes results in some numerical issues i.e., a small fraction of nondominated points may not be discovered in practice.

In light of the above, if we use ES to explore $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$ then $\boldsymbol{z}^{new}$ will be generated. However, in Algorithm 1, it is written that SEARCH($Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$) will return $(\boldsymbol{z}', \boldsymbol{z}'')$. So, it is necessary to mention what $\boldsymbol{z}'$ and $\boldsymbol{z}''$ are. If there is no feasible solution in $\tilde{\mathcal{X}}_E$ that its image in the criterion space lies in the interior of $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$ then the algorithm does as follows:

- For the horizontal cut, the algorithm sets $\boldsymbol{z}' = null$ and $\boldsymbol{z}'' = \boldsymbol{z}^{new}$. This is because there cannot exist any nondominated point in $Box(\bar{\boldsymbol{z}}^1, \boldsymbol{z}^{new})$ in that case.
- For the vertical cut, the algorithm sets $\boldsymbol{z}' = \boldsymbol{z}^{new}$ and $\boldsymbol{z}'' = null$. This is because there cannot exist any nondominated point in $Box(\boldsymbol{z}^{new}, \bar{\boldsymbol{z}}^2)$ in that case.

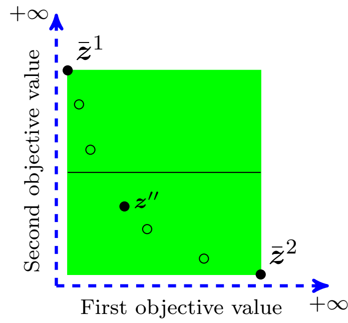
Otherwise, i.e., there exists a feasible solution in $\tilde{\mathcal{X}}_E$ that its image in the criterion space lies in the interior of $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$, the algorithm sets $\boldsymbol{z}' = \boldsymbol{z}'' = \boldsymbol{z}^{new}$. We conclude this section by providing one final comment:

- When using a lexicographic method for finding a nondominated point, it is sometimes possible to avoid solving the second optimization problem. Specifically, when using a horizontal cut, if it turns out that $z_1^{new} = \bar{z}_1^2$ then we must have that $z_2^{new} = \bar{z}_2^2$ (because $\bar{\boldsymbol{z}}^2$ is always a nondominated point by construction). Similarly, when using a vertical cut, if it turns out that $z_2^{new} = \bar{z}_2^1$ then we must have that $z_1^{new} = \bar{z}_1^1$ (because $\bar{\boldsymbol{z}}^1$ is always a nondominated point by construction). So, in such cases, solving the second optimization problem in a lexicographic operation is redundant.

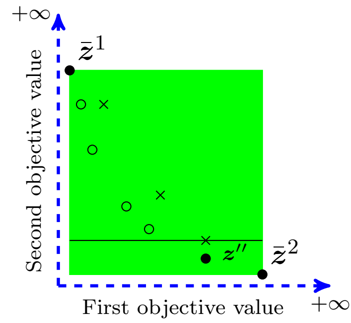
5.2. Balanced box search

Another mechanism available in MSEA.jl is Balanced Box Search (BBS) which is motivated by the balanced box method (Boland et al., 2015). This mechanism basically applies two special cases of ES in sequence. We denote this mechanism by BBS($Direction, Type$). 'Direction' is either horizontal-vertical or vertical-horizontal. For example, horizontal-vertical implies that the first ES that should be called uses a horizontal cut and the second one uses a vertical cut. 'Type' is defined exactly as before (see Section 5.1).

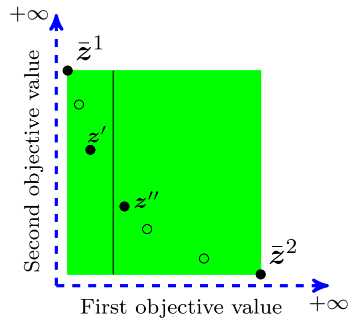
In light of the above, suppose that to explore $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$, we want to use BBS($Horizontal - Vertical, Type$). In that case, the algorithm calls two ES operations in sequence. The first one is ES($Horizontal, Type$) and its output is denoted by $\boldsymbol{z}''$ (which is a nondominated point). This ES is similar to the one presented in Section 5.1 and the only difference is the position of the cut line:



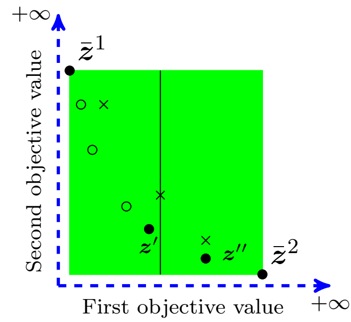
(a) The first ES with no feasible solution



(b) The first ES with feasible solutions



(c) The second ES with no feasible solution



(d) The second ES with feasible solutions

Figure 6: An illustration of the Balanced Box Search for the case with horizontal-vertical direction

- If there exists no feasible solution $\tilde{\boldsymbol{x}} \in \tilde{\mathcal{X}}_E$ such that its image in the criterion space lies in the interior of $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$ then the algorithm will position the horizontal cut exactly in the middle of $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$, i.e., $z_2(\boldsymbol{x}) \leq \frac{\bar{z}_2^1+\bar{z}_2^2}{2}$ (see Figure 6a).
- Otherwise, the algorithm chooses a solution $\tilde{\boldsymbol{x}} \in \tilde{\mathcal{X}}_E$ that its image in the criterion space lies in the interior of $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$ and has the minimum value for the second objective function and then it puts the horizontal cut on $\boldsymbol{z}(\tilde{\boldsymbol{x}})$, i.e., $z_2(\boldsymbol{x}) \leq z_2(\tilde{\boldsymbol{x}})$ (see Figure 6b).

After computing $\boldsymbol{z}''$, ES($Vertical, Type$) should be called for computing a new nondominated point $\boldsymbol{z}'$. This ES is also similar to the one presented in Section 5.1 and the only difference is the position of the cut line:

- If there exists no feasible solution $\tilde{\boldsymbol{x}} \in \tilde{\mathcal{X}}_E$ such that its image in the criterion space lies in the interior of $Box(\bar{\boldsymbol{z}}^1, \boldsymbol{z}'')$ then the algorithm will position the vertical cut exactly on the left side of $\boldsymbol{z}''$, i.e., $z_1(\boldsymbol{x}) \leq z_1'' - \epsilon$ (see Figure 6c).
- Otherwise, the algorithm chooses a solution $\tilde{\boldsymbol{x}} \in \tilde{\mathcal{X}}_E$ that its image in the criterion space lies in the interior of $Box(\bar{\boldsymbol{z}}^1, \boldsymbol{z}'')$ and has the minimum value for the second objective function and then it puts the vertical cut on $\boldsymbol{z}(\tilde{\boldsymbol{x}})$, i.e., $z_1(\boldsymbol{x}) \leq z_1(\tilde{\boldsymbol{x}})$ (see Figure 6d).

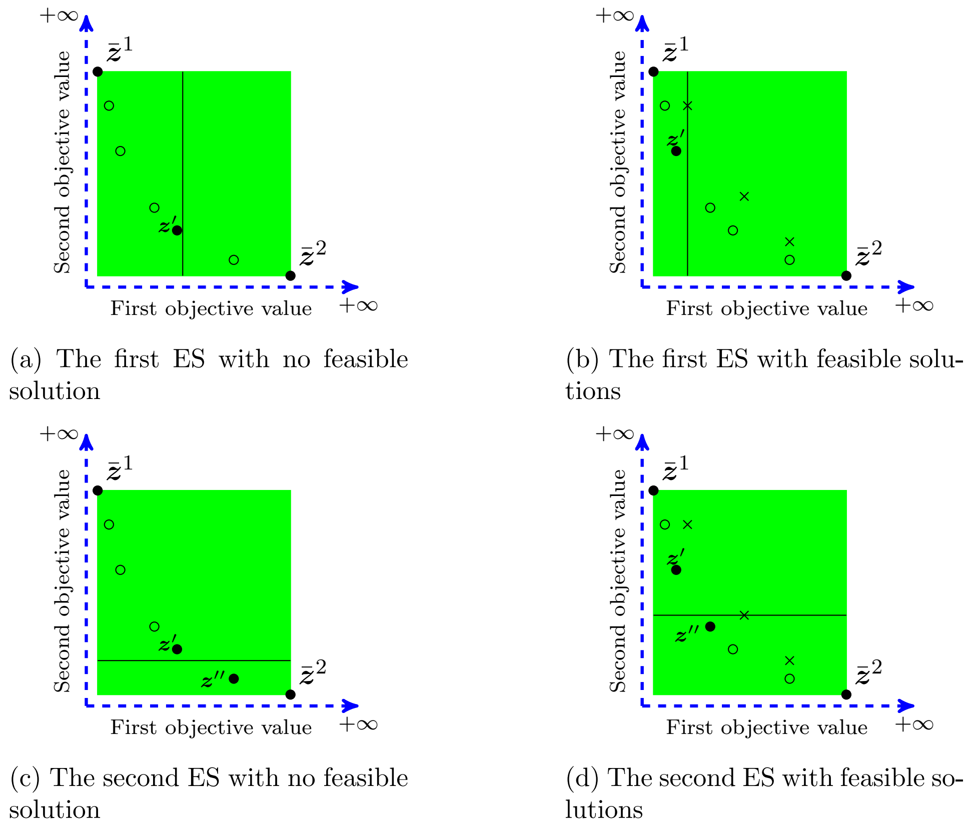
BBS($Vertical - Horizontal, Type$) can be defined similarly. In other words, the algorithm calls two ES operations in sequence. The first one is ES($Vertical, Type$) and its output is denoted by $\boldsymbol{z}'$ (which is a nondominated point). Again, this ES is similar to the one presented in Section 5.1 and the only difference is the position of the cut line:

- If there exists no feasible solution $\tilde{\boldsymbol{x}} \in \tilde{\mathcal{X}}_E$ such that its image in the criterion space lies in the interior of $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$ then the algorithm will position the vertical cut exactly in the middle of $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$, i.e., $z_1(\boldsymbol{x}) \leq \frac{\bar{z}_1^1+\bar{z}_1^2}{2}$ (see Figure 7a).
- Otherwise, the algorithm chooses a solution $\tilde{\boldsymbol{x}} \in \tilde{\mathcal{X}}_E$ that its image in the criterion space lies in the interior of $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$ and has the minimum value for the first objective function and then it puts the vertical cut on $\boldsymbol{z}(\tilde{\boldsymbol{x}})$, i.e., $z_1(\boldsymbol{x}) \leq z_1(\tilde{\boldsymbol{x}})$ (see Figure 7b).

After computing $\boldsymbol{z}'$, ES($Horizontal, Type$) should be called for computing a new nondominated point $\boldsymbol{z}''$. Again, this ES is also similar to the one presented in Section 5.1 and the only difference is the position of the cut line:

- If there exists no feasible solution $\tilde{\boldsymbol{x}} \in \tilde{\mathcal{X}}_E$ such that its image in the criterion space lies in the interior of $Box(\boldsymbol{z}', \boldsymbol{z}^2)$ then the algorithm will position the horizontal cut exactly below $\boldsymbol{z}'$, i.e., $z_2(\boldsymbol{x}) \leq z_2' - \epsilon$ (see Figure 7c).
- Otherwise, the algorithm chooses a solution $\tilde{\boldsymbol{x}} \in \tilde{\mathcal{X}}_E$ that its image in the criterion space lies in the interior of $Box(\boldsymbol{z}', \bar{\boldsymbol{z}}^2)$ and has the minimum value for the first objective function and then it puts the horizontal cut on $\boldsymbol{z}(\tilde{\boldsymbol{x}})$, i.e., $z_2(\boldsymbol{x}) \leq z_2(\tilde{\boldsymbol{x}})$ (see Figure 7d).

We conclude this section by providing one final comment. In Algorithm 1, it is written that if $\boldsymbol{z}' \neq$null and $\boldsymbol{z}'' \neq$null then $Box(\boldsymbol{z}', \boldsymbol{z}'')$ should be added to the priority queue if it may contain some nondominated points. In BBS, $Box(\boldsymbol{z}', \boldsymbol{z}'')$ may contain some not-yet-found nondominated points only if in the second ES, the cut line is positioned on the image of a solution $\tilde{\boldsymbol{x}} \in \tilde{\mathcal{X}}_E$ in the criterion space. In all other cases, by construction, $\boldsymbol{z}'$ and $\boldsymbol{z}''$ must be adjacent nondominated points.



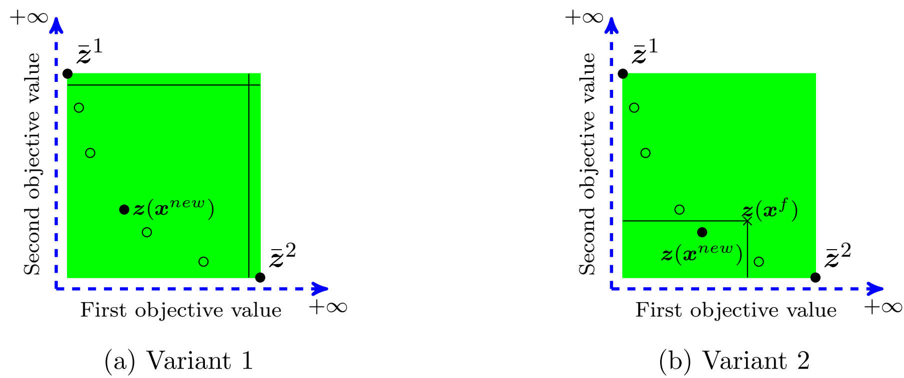
(a) The first ES with no feasible solution

(b) The first ES with feasible solutions

(c) The second ES with no feasible solution

(d) The second ES with feasible solutions

Figure 7: An illustration of the Balanced Box Search for the case with vertical-horizontal direction



(a) Variant 1

(b) Variant 2

Figure 8: An illustration of the Perpendicular search

5.3. Perpendicular search

The last mechanism available in MSEA.jl is Perpendicular Search (PS) which is motivated by the perpendicular search method (Chalmet et al., 1986). This mechanism is denoted by PS($BINS$). 'BINS' or 'Boundary solution Induced Neighborhood Search' is either active or inactive, and it is an effective and simple heuristic for finding a feasible a solution in the interior of $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$ which is proposed by Leitner et al. (2016). PS has three variants:

- **Variant 1:** BINS is not active and no feasible solution is already known in the interior of $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$. In this variant, the following optimization problem should be solved for finding an efficient solution:

$$\boldsymbol{x}^{new} \in \arg\min_{x\in\mathcal{X}}\{z_1(\boldsymbol{x}) + z_2(\boldsymbol{x}) : z_1(\boldsymbol{x}) \leq \bar{z}_1^2 - \epsilon \text{ and } z_2(\boldsymbol{x}) \leq \bar{z}_2^1 - \epsilon\}.$$

  In this case, if this optimization problem is infeasible then we set $\boldsymbol{z}' = \boldsymbol{z}''$ =null. Otherwise, we set $\boldsymbol{z}' = \boldsymbol{z}'' = \boldsymbol{z}(\boldsymbol{x}^{new})$. An illustration of Variant 1 can be found in Figure 8a.

- **Variant 2:** BINS is active and no feasible solution is already known in the interior of $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$. In this variant, the algorithm first attempts to find a feasible solution using BINS. In Algorithm 1, solutions corresponding to top and bottom endpoints, denoted by $\bar{\boldsymbol{x}}^1$ and $\bar{\boldsymbol{x}}^2$, of the box are always stored in the memory. BINS simply fixes each integer decision variable $j \in \{1, \ldots, n\}$ by setting $x_j = \bar{x}_j^1$ if $\bar{x}_j^1 = \bar{x}_j^2$. Obviously, this results in a reduced feasible set denoted by $\mathcal{X}_R$. So, in this case, the following optimization problem will be solved to compute a feasible solution, i.e.,

$$\boldsymbol{x}^f \in \arg\min_{x\in\mathcal{X}_R}\{z_1(\boldsymbol{x}) + z_2(\boldsymbol{x}) : z_1(\boldsymbol{x}) \leq \bar{z}_1^2 - \epsilon \text{ and } z_2(\boldsymbol{x}) \leq \bar{z}_2^1 - \epsilon\}.$$

  If BINS fails to find a feasible solution, i.e., the optimization problem is infeasible, then the algorithm simply calls Variant 1. However, if BINS succeeds then the algorithm finds an efficient solution based on $\boldsymbol{x}^f$ by solving the following optimization problem:

$$\boldsymbol{x}^{new} \in \arg\min_{x\in\mathcal{X}}\{z_1(\boldsymbol{x}) + z_2(\boldsymbol{x}) : z_1(\boldsymbol{x}) \leq z_1(\boldsymbol{x}^f) \text{ and } z_2(\boldsymbol{x}) \leq z_2(\boldsymbol{x}^f)\}.$$

  After computing such an efficient solution, we set $\boldsymbol{z}' = \boldsymbol{z}'' = \boldsymbol{z}(\boldsymbol{x}^{new})$. An illustration of Variant 2 can be found in Figure 8a. Note that $\boldsymbol{x}^f$ is a feasible solution for the above optimization problem and so it can be used to warm start it.

- **Variant 3:** A feasible solution is already known in the interior of $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$. This implies that there are some feasible solutions in $\tilde{\mathcal{X}}_E$ that their images in the criterion space lie in the interior of $Box(\bar{\boldsymbol{z}}^1, \bar{\boldsymbol{z}}^2)$. In this case, the algorithm chooses a $\tilde{\boldsymbol{x}} \in \tilde{\mathcal{X}}_E$ that has the minimum value for the summation of the first and second objective functions. Next, regardless of whether BINS is active or not, the algorithm solves the following optimization problem which is similar to the ones used in Variant 2:

$$\boldsymbol{x}^{new} \in \arg\min_{x\in\mathcal{X}}\{z_1(\boldsymbol{x}) + z_2(\boldsymbol{x}) : z_1(\boldsymbol{x}) \leq z_1(\tilde{\boldsymbol{x}}) \text{ and } z_2(\boldsymbol{x}) \leq z_2(\tilde{\boldsymbol{x}})\}.$$

  After computing $\boldsymbol{x}^{new}$, which is an efficient solution, the algorithm sets $\boldsymbol{z}' = \boldsymbol{z}'' = \boldsymbol{z}(\boldsymbol{x}^{new})$. Note that $\tilde{\boldsymbol{x}}$ is a feasible solution for the above optimization problem and so it can be used to warm start it.

6. Alternating between search mechanisms

MSEA.jl enables users to provide a finite list of the names of search mechanisms for exploring boxes. The length of the list can be arbitrarily large and can be, for example, as follows:

$$\big[\,\text{ES}(\textit{horizontal}, AM); \text{PS}(\textit{inactive}); \ldots; \text{BBS}(\textit{vertical} - \textit{horizontal}, LM2)\big].$$

Each processor $k \in \{1, \ldots, K\}$ will use the list independently to explore the boxes arising during the course of the algorithm. To explore the list, two ways, including *deterministic switching* and *non-deterministic switching*, are implemented in MSEA.jl and users have the option to choose only one of them.

6.1. Deterministic switching

In the deterministic switching, the order of search mechanisms provided in the list is crucial. A processor $k \in \{1, \ldots, K\}$ explores all the boxes in its corresponding priority queue, i.e., Q_k, using the search mechanisms given in the list in the same order. This implies that processor k explores the boxes in Q_k using the first search mechanism available in the list until it finds an empty box. The processor then switches to the second search mechanism available in the list and explores the boxes in Q_k using this mechanism until it finds an empty box. It then switches to the third mechanism available in the list and this procedure repeats. In the case of reaching the last search mechanism in the list, no further switching will be done. Hence, all the remaining boxes of the queue will be explored by the last search mechanism in the list.

6.2. Non-deterministic switching

In the non-deterministic switching, the order of search mechanisms provided in the list is not important. Consequently, the algorithm deals only with the unique search mechanisms given in the list. In other words, if for example, $\text{ES}(\textit{horizontal}, AM)$ is appeared twice in the list then one of them will be removed by the algorithm.

In light of the above, a processor $k \in \{1, \ldots, K\}$ records the time taken by each search mechanism (of the list) in the last time called by the processor. At the first iteration for the processor k, the time is set to zero for all search mechanisms of the list. In each iteration, processor k chooses the search mechanism that has the minimum recored time and applies that to explore the box. As the final comment, in the non-deterministic switching, BBS (and its variants) are not allowed. So, they will be removed from the list if users have provided them. One reason is that BBS is designed to generate at most two nondominated points but the other search mechanisms can generate at most one nondominated point. So, comparing the time of BBS with other mechanism does not make sense. The other reason is that as mentioned before, BBS simply involves applying two specific cases of ES in sequence.

7. Computational study

To evaluate the performance of the proposed method, we conduct a comprehensive computational study. We use the Julia programming language to implement the proposed approach, and employ CPLEX 12.7 as the single-objective linear programming solver. All computational experiments are carried out on a Dell PowerEdge R630 with two Intel Xeon E5-2650 2.2 GHz 12-Core Processors (30MB), 128GB RAM, operating on a RedHat Enterprise Linux 6.8 operating system.

It is worth mentioning that MSEA.jl and the instances are available as open source Julia packages at `https://goo.gl/fbDyCa` and `https://goo.gl/mevqbk`, respectively. The Julia package is compatible with the popular JuMP modeling language (Dunning et al. (2017); Lubin and Dunning (2015)), and supports input in LP and MPS file formats.

Table 1: The list of existing instances in the relevant literature used in this study

Problem	#Constraints	#Variables	#Instances	Source(s)
Knapsack problem	1	50	27	Gandibleux and Freville (2000), Degoutin and Gandibleux (2002), Captivo et al. (2003),Soylu (2015)
		100	6	
		150	5	
		200		
		250		
		300		
		350		
		400		
		450		
		500		
		600	1	
		700		
		800		
		900		
		1,000		
	2	375	5	Boland et al. (2015)
		500		
		625		
		750		
Assignment problem	400	40,000	10	
	600	90,000		
Set covering problem	10	100	4	Soylu (2015), Gandibleux et al. (1998)
	40	200	8	
		400	4	
	60	600	8	
	80	800		
	100	1,000	4	
Set packing problem	300	100	24	Delorme et al. (2010)
	500			
	600	200	36	
	1,000			

We found a total of 274 instances corresponding to different BOPILPs in the relevant literature that are used frequently in multiple studies. Some necessary information about these instances such as the size and some studies that have used them can be found in Table 1. In this computational study, we have considered 10 settings for MSEA.jl in which the list of them can be found in Table 2. Also, it is worth mentioning that a time limit of 5 hours is imposed to solve each instance for each setting.

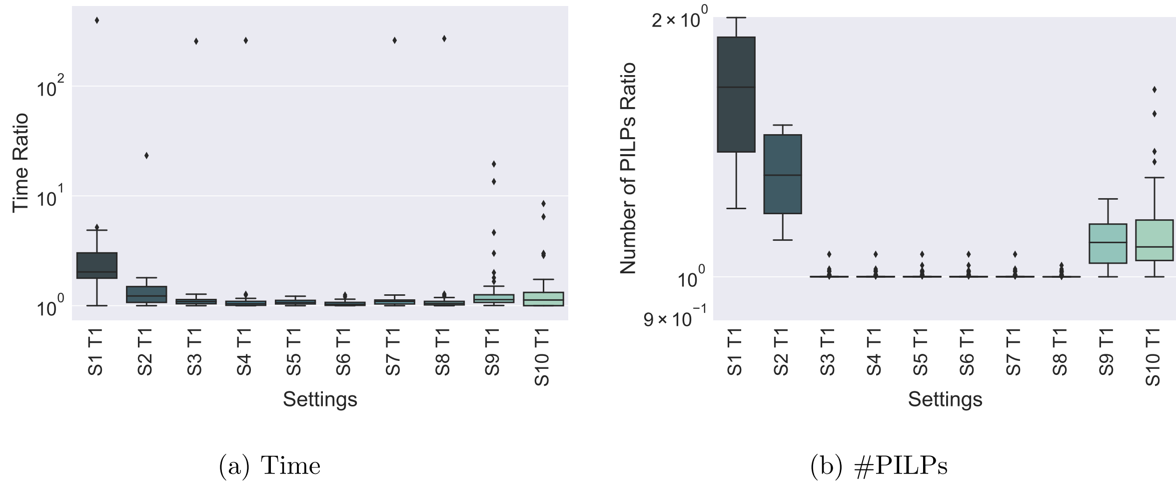
7.1. Single thread

In this section, it is assumed that only one thread is available. So, the notation $T1$ is used to show this fact. The computational results are reported using box plots. Overall, two types of box plots are employed: one is used for comparing the *normalized* number of PILPs solved and the other is used for comparing the *normalized* solution time. To explain the term "normalized", suppose that we want to compare a subset of settings of MSEA.jl in terms of solution time for a subset of the instances. In this case, for each instance (of the subset), we first compute the minimum solution time among all settings (in the subset). Next, the solution time of each setting for each instance will be divided by the obtained minimum solution time and then the box plots will be

Table 2: Different settings of MSEA.jl used in this study

Setting	List of Search Mechanisms	Phase I	Switch
S1	$\big[\text{PS}(active)\big]$	inactive	deterministic
S2	$\big[\text{BBS}(horizontal - vertical, LM1)\big]$	inactive	deterministic
S3	$\big[\text{ES}(horizontal, LM1)\big]$	inactive	deterministic
S4	$\big[\text{ES}(horizontal, LM2)\big]$	inactive	deterministic
S5	$\big[\text{ES}(vertical, LM1)\big]$	inactive	deterministic
S6	$\big[\text{ES}(vertical, LM2)\big]$	inactive	deterministic
S7	$[S3; S4; S5; S6]$	inactive	deterministic
S8	$[S3; S4; S5; S6]$	inactive	non-deterministic
S9	$[S3; S4; S5; S6]$	FPBH.jl	non-deterministic
S10	$[S1; S3; S4; S5; S6]$	FPBH.jl	non-deterministic

created accordingly. A similar process will be done for creating box plots based on the number of PILPs solved.



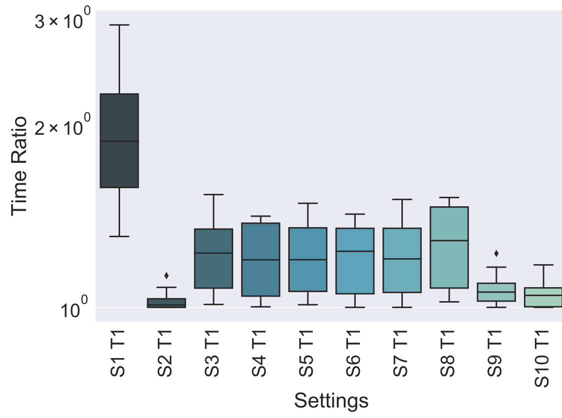
(a) Time

(b) #PILPs

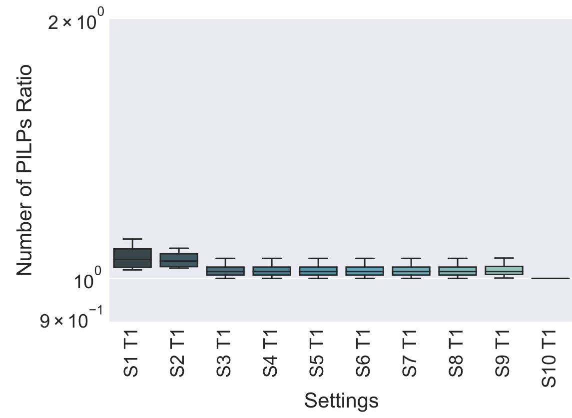
Figure 9: Comparing all 10 settings for all instances of the knapsack problem

In light of the above, the box plots for the (normalized) solution time and the (normalized) number of PILPs solved for all instances of the knapsack problem, assignment problem, set covering problem, and set packing problem can be found in Figures 9-12, respectively. For the knapsack problem, settings S3-S8 perform well. However, S6 performs marginally better in terms of the solution time and in terms of the number of SOILPs solved, S8 performs slightly better. Observe that S1 performs the worst in terms of the solution time for this class of optimization problems because its solution time is around 5 times worse than the best case scenario on average.

For the assignment problem, S10 solves the minimum number of PILPs but the S2 has the least solution time on average. Also, again S1 performs the worse in terms of the solution time. However, for the set covering problem, S4, performs slightly better than other settings while S3-S8 solve the minimum number of PILPs. Finally, for set packing instances, S4 and S7 seem to be the best in terms of the solution time while again S3-S8 solve the minimum number of PILPs.

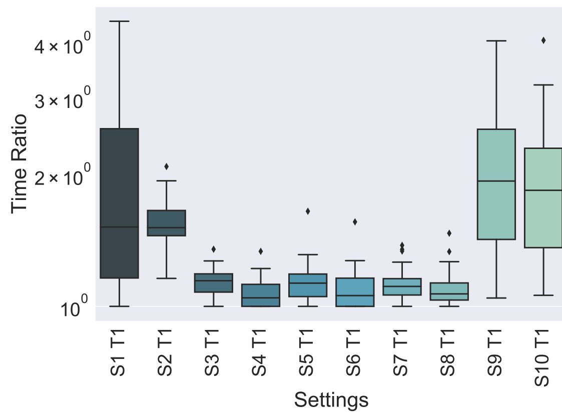


(a) Time

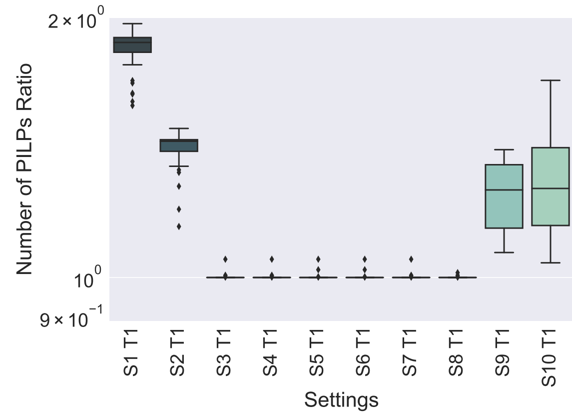


(b) #PILPs

Figure 10: Comparing all 10 settings for all instances of the assignment problem

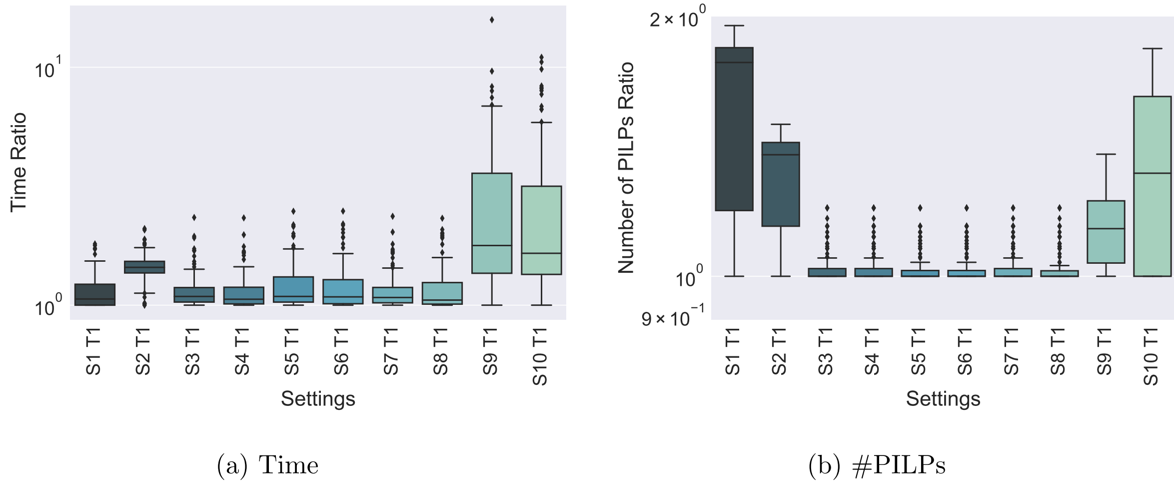


(a) Time



(b) #PILPs

Figure 11: Comparing all 10 settings for all instances of the set covering problem

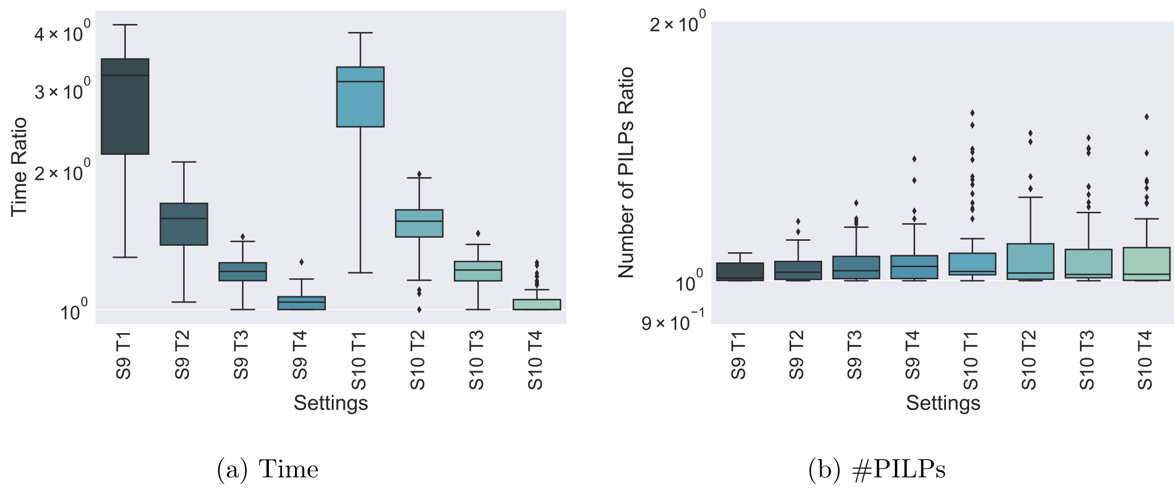


(a) Time (b) #PILPs

Figure 12: Comparing all 10 settings for all instances of the set packing problem

7.2. Multiple threads

In this section, it is assumed that the number of threads can be between 1 to 4, i.e, $T1$, $T2$, $T3$, and $T4$. The box plots corresponding to settings $S9$ and $S10$ for large instances, i.e., those that take at least 600 seconds to be solved when using a single thread, can be found in Figure 13. Observe that for both settings, the solution time decreases dramatically as the number of threads increases. When there are four threads available, the solution time has decreased by a factor of more than 3 on average. Observe that the number of SOILPs is different when different number of processors are available for each setting. The reason for this observation is that (as we discussed in Section 6) each processor decides about switching between search mechanisms independently.



(a) Time (b) #PILPs

Figure 13: Comparing settings 9 and 10 for all large instances when multiple threads are available

8. Final remarks

We developed a framework/algorithm, i.e., MSEA, for combining many variants of several criterion space search algorithms in the literature of bi-objective pure integer linear programming. MSEA decomposed the criterion space into boxes and enables users to search them by variants of several existing search operations in the literature. So, MSEA has the potential to be used by researchers and practitioners as a research platform for creating custom-built algorithms for their specific problems. MSEA can generate both exact and approximate nondominated frontiers and it employs some internal heuristic approaches. Finally, MSEA supports execution on multiple processors and the computational study showed that this can reduce the solution time significantly in practice. It is worth mentioning that Julia implementation of the proposed framework named MSEA.jl is publicly available in GitHub. It is expected that the algorithm to be extended further and future versions of MSEA.jl include more features and search operations, for example Tchebycheff operation (Dächert et al., 2012). Also, MSEA.jl currently uses CPLEX for solving single-objective optimization problems. So, it is expected that in the future versions more options (for example Gurobi, SCIP, GLPK) to be available for users. Finally, it is worth mentioning that extending the proposed framework for pure integer linear programs with more than two objectives is a future research direction that is worth to explore.

References

Boland, N., Charkhgard, H., Savelsbergh, M., 2015. A criterion space search algorithm for biobjective integer programming: The balanced box method. INFORMS Journal on Computing 27 (4), 735–754.

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.

Captivo, M. E., Clımaco, J., Figueira, J., Martins, E., Santos, J. L., 2003. Solving bicriteria 01 knapsack problems using a labeling algorithm. Computers & Operations Research 30 (12), 1865 – 1886.

Chalmet, L. G., Lemonidis, L., Elzinga, D. J., 1986. An algorithm for bi-criterion integer programming problem. European Journal of Operational Research 25, 292–300.

Chankong, V., Haimes, Y. Y., 1983. Multiobjective Decision Making: Theory and Methodology. Elsevier Science, New York.

Dächert, K., Gorski, J., Klamroth, K., 2012. An augmented weighted Tchebycheff method with adaptively chosen parameters for discrete bicriteria optimization problems. Computers & Operations Research 39, 2929–2943.

Dai, R., Charkhgard, H., 2018. A two-stage approach for bi-objective integer linear programming. Operations Research Letters 46 (1), 81 – 87.

Degoutin, F., Gandibleux, X., 2002. Un retour dexpériences sur la résolution de problemes combinatoires bi-objectifs.

Delorme, X., Gandibleux, X., Degoutin, F., 2010. Evolutionary, constructive and hybrid procedures for the bi-objective set packing problem. European Journal of Operational Research 204 (2), 206 – 217.

Dunning, I., Huchette, J., Lubin, M., 2017. Jump: A modeling language for mathematical optimization. SIAM Review 59 (2), 295–320.

Dchert, K., Klamroth, K., Lacour, R., Vanderpooten, D., 2017. Efficient computation of the search region in multi-objective optimization. European Journal of Operational Research 260 (3), 841 – 855.

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

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.

Gandibleux, X., Freville, A., 2000. Tabu search based procedure for solving the 0-1 multiobjective knapsack problem: The two objectives case. Journal of Heuristics 6 (3), 361–383.

Gandibleux, X., Vancoppenolle, D., Tuyttens, D., 1998. A first making use of GRASP for solving MOCO problems. Tech. rep., University of Valenciennes, France.

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.

Leitner, M., Ljubić, I., Sinnl, M., Werner, A., 2016. ILP heuristics and a new exact method for bi-objective 0/1 ILPs: Application to FTTx-network design. Computers & Operations Research 72, 128–146.

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

Lubin, M., Dunning, I., 2015. Computing in operations research using julia. INFORMS Journal on Computing 27 (2), 238–248.

Özlen, M., Azizoğlu, M., 2009. Multi-objective integer programming: A general approach for generating all non-dominated solutions. European Journal of Operational Research 199, 25–35.

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.

Pal, A., Charkhgard, H., 2018a. A feasibility pump and local search based heuristic for bi-objective pure integer linear programming. INFORMS Journal On Computing. To appear.

Pal, A., Charkhgard, H., 2018b. Fpbh.jl: A feasibility pump based heuristic for multi-objective mixed integer linear programming in julia. Preprint. http://www.optimization-online.org/DB_FILE/2017/09/6195.pdf.

Pedroso, D. M., Bonyadi, M. R., Gallagher, M., 2017. Parallel evolutionary algorithm for single and multi-objective optimisation: Differential evolution and constraints handling. Applied Soft Computing 61, 995 – 1012.

Pettersson, W., Özlen, M., 2017. A parallel approach to bi-objective integer programming. ANZIAM Journal 58 (0), 69–81.

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

Soylu, B., 2015. Heuristic approaches for biobjective mixed 0-1 integer linear programming problems. European Journal of Operational Research 245 (3), 690 – 703.

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

Stidsen, T., Andersen, K. A., Dammann, B., 2014. A branch and bound algorithm for a class of biobjective mixed integer programs. Management Science 60 (4), 1009–1032.

Tricoire, F., 2012. Multi-directional local search. Computers & Operations Research 39 (12), 3089 – 3101.

Yu, W.-J., Li, J.-Z., Chen, W.-N., Zhang, J., 2017. A parallel double-level multiobjective evolutionary algorithm for robust optimization. Applied Soft Computing 59, 258 – 275.