FPBH.jl: A Feasibility Pump Based Heuristic for Multi-objective Mixed Integer Linear Programming in Julia

Aritra Pal and Hadi Charkhgard

Department of Industrial and Management System Engineering, University of South Florida, Tampa, FL, 33620 USA

September 8, 2017

Abstract

Feasibility pump is one of the successful heuristic solution approaches developed almost a decade ago for computing high-quality feasible solutions of single-objective integer linear programs, and it is implemented in exact commercial solvers such as CPLEX and Gurobi. In this study, we present the first Feasibility Pump Based Heuristic (FPBH) approach for approximately generating nondominated frontiers of multi-objective mixed integer linear programs with an arbitrary number of objective functions. The proposed algorithm extends our recent study for bi-objective pure integer programs that employs a customized version of several existing algorithms in the literature of both single-objective and multi-objective optimization. The method has two desirable characteristics: (1) There is no parameter to be tuned by users other than the time limit; (2) It can naturally exploit parallelism. An extensive computational study shows the efficacy of the proposed method on some existing standard test instances in which the true frontier is known, and also some randomly generated instances. We also numerically show the importance of parallelization feature of FPBH and illustrate that FPBH outperforms MDLS developed by Tricoire [39] on instances of multi-objective (1-dimensional) knapsack problem. We test the effect of using different commercial and non-commercial linear programming solvers for solving linear programs arising during the course of FPBH, and show that the performance of FPBH is almost the same in all cases. It is worth mentioning that FPBH is available as an open source Julia package, named as 'FPBH.jl', in GitHub. The package is compatible with the popular JuMP modeling language (Dunning et al. [16] and Lubin and Dunning [27]), supports input in LP and MPS file formats. The package can plot nondominated frontiers, can compute different quality measures (hypervolume, cardinality, coverage and uniformity), supports execution on multiple processors and can use any linear programming solver supported by MathProgBase.jl (such as CPLEX, Clp, GLPK, etc).

Keywords. Multi-objective mixed integer linear programming, feasibility pump, local search, local branching, parallelization

1 Introduction

Multi-objective optimization provides decision-makers with a complete view of the trade-offs between their objective functions that are attainable by feasible solutions. It is thus a critical tool in engineering and management, where competing goals must often be considered and balanced when making decisions. For example, in business settings, there may be trade-offs between long-term profits and short-term cash flows or between cost and reliability, while in public good settings, there can be trade-offs between providing benefits to different communities or between environmental impacts and social good. Especially when such trade-offs are difficult to make, enabling decision-makers to see the range of possibilities offered by the nondominated frontier (or a portion of the nondominated frontier), before selecting their preferred solution, can be highly valuable. Combined with the fact that many engineering problems can be formulated as mixed integer linear programs, the development of efficient and reliable multi-objective mixed integer linear programming solvers may have significant benefits for problem solving in industry and government.

In recent years, several studies have been conducted on developing exact algorithms for solving multi-objective optimization problems, see for instance Dächert et al. [13], Dächert and Klamroth [14], Kirlik and Sayın [23], Köksalan and Lokman [24], Özlen et al. [29], Przybylski and Gandibleux [33], Przybylski et al. [34], Soylu and Yıldız [38], and Boland et al. [5, 6, 7]. This resurgence of interest on exact solution approaches for solving multi-objective optimization problems in the last decade is promising. However, there is still a large proportion of multi-objective optimization problems that either no exact method is known for them (such as mixed integer linear programs with more than two objective functions) or exact solutions approaches cannot solve them in reasonable time (for example, large size instances of NP-hard problems).

Consequently, developing heuristic solution approaches for multi-objective optimization problems is worth studying. We note that in the last two decades, significant advances have been made on developing evolutionary methods for solving multi-objective optimization problems. A website maintained by Carlos A. Coello Coello (`http://delta.cs.cinvestav.mx/~ccoello/EMOO/`), for example, lists close to 4,700 journal papers and around 3,800 conference papers on the topic (inducing but not limited to [10, 12, 15, 19, 25, 28, 40]).

Surprisingly, most of these methods are not specifically designed for problems with integer decision variables. Even if they can directly handle integer decision variables, they are often problem-dependent algorithms. This means that they are usually designed for solving a certain class of optimization problems, for example, multi-objective facility location problem. Moreover, many of the existing generic heuristics are designed to solve unconstrained multi-objective optimization problems. So, if there are some constraints in the problem, these methods will usually remove them by penalizing them and adding them to the objective functions.

In light of the above, the literature of multi-objective mixed integer programming is suffering from the lack of studies on generic heuristic approaches. Three of very few studies in this scope are conducted by Tricoire [39], Lian et al. [26] and Paquete et al. [32]. In these studies, two successful heuristics, the so-called Multi-Directional Local Search (MDLS) and Pareto Local Search (PLS), are developed that can solve multi-objective pure integer linear programs. We note that, these methods have been shown to be successful in practice, but they have two main weaknesses: (1) It is not clear whether they can handle mixed integer linear programs with multiple objectives since they are not specifically designed to do so; (2) Users have to customize these algorithms to be able to solve their specific problems. In other words, only the framework of these algorithms is generic.

In this study, we develop an algorithm that overcomes both of these weaknesses. The proposed heuristic is generic, easy-to-use, and can approximate the nondominated frontier of any multi-objective mixed (or pure) integer linear program with an arbitrary number of objective functions. The proposed method has two other desirable characteristics as well. (1) There is no parameter to be tuned by users other than the time limit. (2) It can naturally exploit parallelism.

The engine of the proposed algorithm is basically the well-known *feasibility pump* heuristic [3, 20, 1, 8, 22]. The feasibility pump is one of the successful heuristic solution approaches that developed just a decade ago for solving single-objective mixed integer linear programs. The approach is so successful and easy-to-implement that quickly after its invention it found its way into commercial optimization solvers such as CPLEX and Gurobi. Consequently, the main motivation of our study is that developing a similar technique for multi-objective mixed integer linear programs may follow the same path in the future.

This is highlighted by the fact in our recent study, we developed a feasibility pump based heuristic for solving bi-objective pure integer linear programs (see Pal and Charkhgard [31]), and obtain promising computational results. So, in this study, we extend our previous work to multi-objective mixed integer linear programs with an arbitrary number of objective functions. The algorithm developed, in our previous study, combines the underlying ideas of several exact/heuristic algorithms in the literature of both single and multi-objective optimization including the perpendicular search method [11, 4], a local search approach [39], and the weighted sum method [2] as well as the feasibility pump. Although our new method inherit the key components of our previous study, but it is completely different and novel. The following are some of the main differences:

- In the new method, the underlying idea of another well-known technique, the so-called *local branching* [21], is also added to the algorithm. This is motivated by the observation that local branching can align well with the feasibility pump for single-objective optimization problems [36]. In this study, we show that the same observation is true for multi-objective optimization problems as well.

- The weighted sum method used in the previous study does not work for instances with more than two objectives. So, we develop a different weighted sum method for those instances.

- The underlying decomposition technique of the perpendicular search method used in the previous study does not work for problems with more than two objectives. So, in this study, we employ a different but effective decomposition technique that does not dependent on the number of objective functions.

- The local search method developed in the previous study is used in this study as well. However, in addition to that, we develop a new local search technique that should be activated if there exist continuous variables in an instance.

We conduct an extensive computational study that has three parts. In the first part, we test our algorithm on bi-objective mixed integer linear programs. In the second part, the performance of the new algorithm is tested on multi-objective pure integer programs. Finally, in the third part, a set of experiments is conducted to show the performance of the algorithm on mixed integer linear programs with more than two objectives. In each of these parts, the importance of different components of the algorithm as well as its parallelization feature are shown. Furthermore, in each part, the effect

of using different linear programming solvers, including CPLEX, GUROBI, SCIP, GLPK and Clp, at the heart of the proposed algorithm is illustrated. It is shown that the proposed algorithm performs almost the same with any of these solvers. Finally, since the implementation of MDLS developed for (1-dimensional) knapsack problem is publicly available, we compare the performance of our algorithm against MDLS on some existing instances for this class of optimization problem, and show that our algorithm performs better.

The remainder of the paper is organized as follows. In Section 2, we introduce notation and some fundamental concepts. In Section 3, we provide the general framework of our algorithm. In Section 4, we introduce the proposed weighted sum operation. In Section 5, we introduce the proposed feasibility pump operation. In Section 6, the proposed local search operation is explained. In Section 7, we conduct an extensive computational study. Finally, in Section 8, we give some concluding remarks.

2 Preliminaries

A *multi-objective mixed integer linear program* (MOMILP) can be stated as follows:

$$\min_{(\boldsymbol{x}_1,\boldsymbol{x}_2)\in\mathcal{X}} \{z_1(\boldsymbol{x}_1,\boldsymbol{x}_2),\dots,z_2(\boldsymbol{x}_1,\boldsymbol{x}_2)\}, \tag{1}$$

where $\mathcal{X} := \left\{(\boldsymbol{x}_1,\boldsymbol{x}_2) \in \mathbb{Z}^{n_1}_{\geq} \times \mathbb{R}^{n_2}_{\geq} : A_1\boldsymbol{x}_1 + A_2\boldsymbol{x}_2 \leq \boldsymbol{b}\right\}$ represents the *feasible set in the decision space*, $\mathbb{Z}^{n_1}_{\geq} := \{\boldsymbol{s} \in \mathbb{Z}^{n_1} : \boldsymbol{s} \geq \boldsymbol{0}\}$, $\mathbb{R}^{n_2}_{\geq} := \{\boldsymbol{s} \in \mathbb{R}^{n_2} : \boldsymbol{s} \geq \boldsymbol{0}\}$, $A_1 \in \mathbb{R}^{m\times n_1}$, $A_2 \in \mathbb{R}^{m\times n_2}$, and $\boldsymbol{b} \in \mathbb{R}^m$. It is assumed that $z_i(\boldsymbol{x}_1,\boldsymbol{x}_2) = \boldsymbol{c}_i^\intercal\boldsymbol{x}_1 + \boldsymbol{d}_i^\intercal\boldsymbol{x}_2$ where $\boldsymbol{c}_i \in \mathbb{R}^{n_1}$ and $\boldsymbol{d}_i \in \mathbb{R}^{n_2}$ for $i = 1,\dots,p$ represents a linear objective function. The image $\mathcal{Y}$ of $\mathcal{X}$ under vector-valued function $\boldsymbol{z} := (z_1,\dots,z_p)^\intercal$ represents the *feasible set in the objective/criterion space*, i.e., $\mathcal{Y} := \{\boldsymbol{o} \in \mathbb{R}^2 : \boldsymbol{o} = \boldsymbol{z}(\boldsymbol{x}_1,\boldsymbol{x}_2) \text{ for all } (\boldsymbol{x}_1,\boldsymbol{x}_2) \in \mathcal{X}\}$. We denote the linear programming (LP) relaxation of $\mathcal{X}$ by $LP(\mathcal{X})$, and we assume that it is *bounded*.

Since we assume that $LP(\mathcal{X})$ is bounded, for any integer decision variable $x_{1,j}$ where $j \in \{1,\dots,n_1\}$, there must exist a finite global upper bound u_j. So, it is easy to show that any MOMILP can be reformulated as a MOMILP with only binary decision variables. For example, we can replace any instance of x_{1j} by $\sum_{k=0}^{\lfloor \log_2 u_j \rfloor} 2^k x'_{1,j,k}$ where $x'_{1,j,k} \in \{0,1\}$ in the mathematical formulation. So, in this paper, without loss of generality, we assume that the mathematical formulation contains only binary decision variables, i.e., $\mathcal{X} \subseteq \{0,1\}^{n_1} \times \mathbb{R}^{n_2}$.

Definition 1. A feasible solution $(\boldsymbol{x}_1^I,\boldsymbol{x}_2^I) \in \mathcal{X}$ is called *ideal* if it minimizes all objectives simultaneously, i.e., if $(\boldsymbol{x}_1^I,\boldsymbol{x}_2^I) \in \arg\min_{(\boldsymbol{x}_1,\boldsymbol{x}_2)\in\mathcal{X}} z_i(\boldsymbol{x}_1,\boldsymbol{x}_2)$ for all $i = 1,\dots,p$.

Definition 2. The point $\boldsymbol{z}^I \in \mathbb{R}^p$ is the *ideal point* if $z_i^I = \min_{\boldsymbol{x}\in\mathcal{X}} z_i(\boldsymbol{x}_1^I,\boldsymbol{x}_2^I)$ for all $i \in \{1,\dots,p\}$.

Note that the ideal point $\boldsymbol{z}^I$ is an imaginary point in the criterion space unless an ideal solution exists. Obviously, if an ideal solution $(\boldsymbol{x}_1^I,\boldsymbol{x}_2^I)$ exists then $\boldsymbol{z}^I = \boldsymbol{z}(\boldsymbol{x}_1^I,\boldsymbol{x}_2^I)$. However an ideal solution does not often exist in practice.

Definition 3. A feasible solution $(\boldsymbol{x}_1,\boldsymbol{x}_2) \in \mathcal{X}$ is called *efficient* or *Pareto optimal*, if there is no other $(\boldsymbol{x}'_1,\boldsymbol{x}'_2) \in \mathcal{X}$ such that $z_i(\boldsymbol{x}'_1,\boldsymbol{x}'_2) \leq z_i(\boldsymbol{x}_1,\boldsymbol{x}_2)$ for $i = 1,\dots,p$ and $\boldsymbol{z}(\boldsymbol{x}'_1,\boldsymbol{x}'_2) \neq \boldsymbol{z}(\boldsymbol{x}_1,\boldsymbol{x}_2)$. If $(\boldsymbol{x}_1,\boldsymbol{x}_2)$ is efficient, then $\boldsymbol{z}(\boldsymbol{x}_1,\boldsymbol{x}_2)$ 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*.

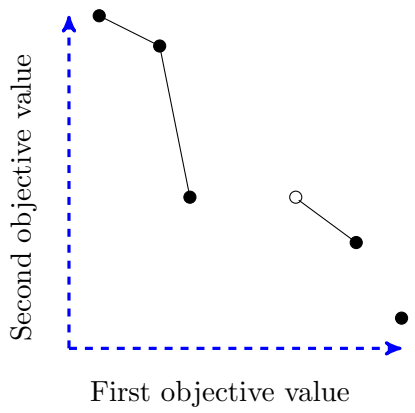


Figure 1: An illustration of the nondominated frontier of a MOMILP with $p = 2$

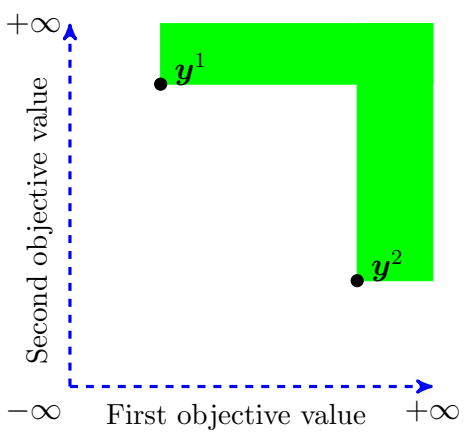
Overall, multi-objective optimization is concerned with finding a full representation of the nondominated frontier. Specifically, $\min_{(\boldsymbol{x}_1,\boldsymbol{x}_2)\in\mathcal{X}} \boldsymbol{z}(\boldsymbol{x}_1,\boldsymbol{x}_2)$ is defined to be precisely $\mathcal{Y}_N$. An illustration of the nondominated frontier of a MOMILP when $p = 2$ is shown in Figure 1 [5]. We observe that solving a MOMILP even when $p = 2$ is quite challenging since its nondominated frontier is not convex and it may contain some points, line segments, and even half-open (or open) line segments. It is evident that for $p > 2$, there may exist (hyper)plane segments in the nondominated frontier as well.

In light of the above, in this study, we develop a heuristic approach, i.e., FPBH, for approximately representing the nondominated frontier of a MOMILP. FPBH simply approximates the nondominated frontier of a MOMILP by generating a finite set of feasible points. Since our algorithm is a heuristic method, there is no guarantee that the set of points generated by the method to be a subset of the exact nondominated points of the problem. However, the goal is that the set to be a high-quality approximation for the exact nondominated frontier in practice. Next, we introduce concepts and notation that will facilitate the presentation and discussion of the proposed approach.

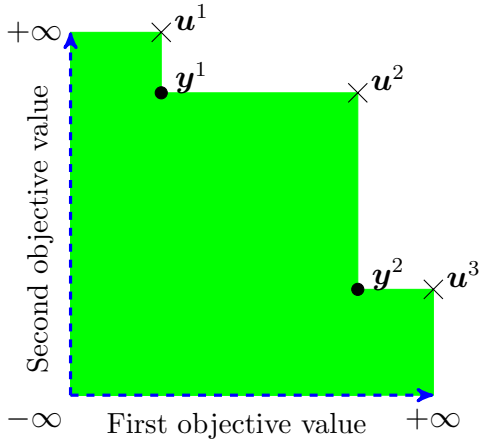
In FPBH, we often deal with a finite subset of feasible solutions, $S \subseteq \mathcal{X}$, of which we would like to remove dominated solutions (those that are dominated by other feasible solutions of S). We denote this operation by $\textsc{Pareto}(S)$, and it simply returns the following set,

$$\{(\boldsymbol{x}_1,\boldsymbol{x}_2) \in \mathcal{S} : \nexists (\boldsymbol{x}'_1,\boldsymbol{x}'_2) \in \mathcal{S} \text{ with } z_i(\boldsymbol{x}'_1,\boldsymbol{x}'_2) \leq z_i(\boldsymbol{x}_1,\boldsymbol{x}_2) \text{ for } i = 1,\dots,p \ , \boldsymbol{z}(\boldsymbol{x}'_1,\boldsymbol{x}'_2) \neq \boldsymbol{z}(\boldsymbol{x}_1,\boldsymbol{x}_2)\}.$$

Note that in this operation, we make sure that the created set is *minimal* in a sense that if there are multiple solutions with the same objective values then only one of them can be in the set. Let $S \subseteq \mathcal{X}$ be a finite subset of feasible solutions. Obviously, $\cup_{(\boldsymbol{x}_1,\boldsymbol{x}_2)\in S}\{\boldsymbol{z}(\boldsymbol{x}_1,\boldsymbol{x}_2) + \mathbb{R}^p_{\geq}\}$ is the set of all points of the criterion space dominated by solutions in set S (of course the only exceptions are the points in set $\cup_{(\boldsymbol{x}_1,\boldsymbol{x}_2)\in S}\{\boldsymbol{z}(\boldsymbol{x}_1,\boldsymbol{x}_2)\}$). For example, suppose that $S = \{(\boldsymbol{x}^1_1,\boldsymbol{x}^1_2),(\boldsymbol{x}^2_1,\boldsymbol{x}^2_2)\}$ and let $\boldsymbol{y}^1 := \boldsymbol{z}(\boldsymbol{x}^1_1,\boldsymbol{x}^1_2)$ and $\boldsymbol{y}^2 := \boldsymbol{z}(\boldsymbol{x}^2_1,\boldsymbol{x}^2_2)$. An illustration of the feasible points $\boldsymbol{y}^1$ and $\boldsymbol{y}^2$ and $\cup_{(\boldsymbol{x}_1,\boldsymbol{x}_2)\in S}\{\boldsymbol{z}(\boldsymbol{x}_1,\boldsymbol{x}_2) + \mathbb{R}^p_{\geq}\}$ when $p = 2$ can be found in Figure 2a. Kirlik and Sayın [23] introduce a simple recursive algorithm for finding the *minimum* set of points denoted by $\boldsymbol{u}^1,\dots,\boldsymbol{u}^K$ with the property that $\cup_{k=1}^K\{\boldsymbol{u}^k - \mathbb{R}^p_{>}\}$ defines the set of all points in the criterion space that are not dominated by solutions in S. We call $\boldsymbol{u}^1,\dots,\boldsymbol{u}^K$ as the set of upper bound points, and use the operation $\textsc{Decompose}(S)$ to compute them. An illustration of the upper bound points corresponding to $\boldsymbol{y}^1$ and $\boldsymbol{y}^2$, and also $\cup_{k=1}^K\{\boldsymbol{u}^k - \mathbb{R}^p_{>}\}$ when $p = 2$ can be found in Figure 2b. We next briefly explain how the recursive algorithm works.



(a) Dominated regions by $\boldsymbol{y}^1$ and $\boldsymbol{y}^2$



(b) Regions not dominated by $\boldsymbol{y}^1$ and $\boldsymbol{y}^2$

Figure 2: An illustration of the upper bound points when $p = 2$ and two feasible points $\boldsymbol{y}^1$ and $\boldsymbol{y}^2$ are known

Let $S = S' \cup \{(\boldsymbol{x}_1, \boldsymbol{x}_2)\}$ and $|S| = |S'| + 1$. Now, suppose that the minimum set of upper bound points corresponding to set S', denoted by $\boldsymbol{u}^1, \ldots, \boldsymbol{u}^K$, is known. As an aside, if $S' = \emptyset$ then we consider $+\boldsymbol{\infty} := (+\infty, \ldots, +\infty)$ as the only upper bound point of the set S'. Now, to compute the minimum set of upper bound points corresponding to set S, two steps should be taken:

- **Step 1:** For each $k \in \{1, \ldots, K\}$ with $\boldsymbol{z}(\boldsymbol{x}_1, \boldsymbol{x}_2) \leq \boldsymbol{u}^k$ and $\boldsymbol{z}(\boldsymbol{x}_1, \boldsymbol{x}_2) \neq \boldsymbol{u}^k$, we replace $\boldsymbol{u}^k$ by p new upper bound points denoted by $\boldsymbol{u}^{k,1}, \ldots, \boldsymbol{u}^{k,p}$. For each $i = 1, \ldots, p$, we set $u_i^{k,i} = z_i(\boldsymbol{x}_1, \boldsymbol{x}_2)$ and $u_l^{k,i} = u_l^k$ for each $l \in \{1, \ldots, p\} \backslash \{i\}$.
- **Step 2:** *Redundant* elements of the set generated in Step 1 should be removed one by one in this step. An upper bound $\boldsymbol{u}^v$ is redundant if there exists another upper bound point $\boldsymbol{u}^w$ such that $\boldsymbol{u}^v \leq \boldsymbol{u}^w$.

Interested readers may refer to the study of Kirlik and Sayın [23] for further details about the recursive method for generating the upper bound points. It is worth mentioning that there are more efficient but more complicated approach in order to compute the upper bound points (see for instance [17]). However, in this study, we use the simple approach explained above since it is easier to implement (and performs well in practice).

3 The framework of the proposed algorithm

The algorithm is a two-stage approach and it maintains a list of all $\boldsymbol{x}_1 \in \{0,1\}^{n_1}$ that based on which the algorithm was not able to obtain a feasible solution. We call this list as *Tabu* and at the beginning of the algorithm, this list is empty. The algorithm uses this list in all feasibility pump operations to avoid cycles. Since the size of the *Tabu* list becomes large during the course of the first stage, we make this list empty before starting the second stage to save the valuable computational time. The algorithm also maintains a set of feasible solutions, denoted by $\tilde{\mathcal{X}}$, during the course of

Algorithm 1: The framework of the Algorithm

$List.create(Tabu)$
$\tilde{\mathcal{X}} \leftarrow 0$
$Time\_Limit\_Stage1 \leftarrow \gamma \times Total\_Time\_Limit$
$Time\_Limit\_Stage2 \leftarrow (1-\gamma) \times Total\_Time\_Limit$
$(\tilde{\mathcal{X}}, Tabu) \leftarrow Stage1\_Algorithm\big(\tilde{\mathcal{X}}, Tabu, Time\_Limit\_Stage1\big)$
$List.EraseElements(Tabu)$
$\tilde{\mathcal{X}} \leftarrow Stage2\_Algorithm(\tilde{\mathcal{X}}, Tabu, Time\_Limit\_Stage2)$
return $\tilde{\mathcal{X}}$

the algorithm. When the algorithm terminates this set will be reported as the set of approximate efficient solutions of the problem. In other words, when the algorithm terminates any two distinct solutions $(\boldsymbol{x}_1, \boldsymbol{x}_2), (\boldsymbol{x}'_1, \boldsymbol{x}'_2) \in \tilde{\mathcal{X}}$ do not dominate each other, i.e., there exist $i, j \in \{1, \ldots, p\}$ such that $z_i(\boldsymbol{x}_1, \boldsymbol{x}_2) > z_i(\boldsymbol{x}'_1, \boldsymbol{x}'_2)$ and $z_j(\boldsymbol{x}_1, \boldsymbol{x}_2) < z_j(\boldsymbol{x}'_1, \boldsymbol{x}'_2)$.

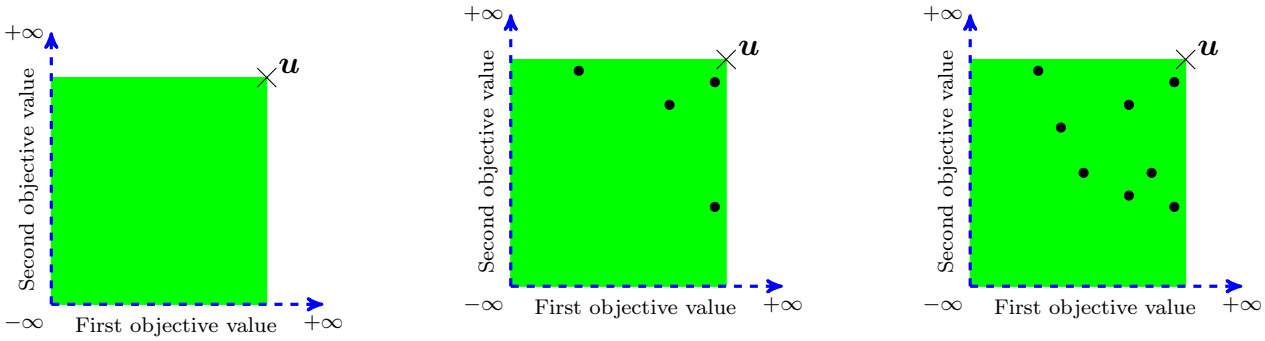
The algorithm takes two inputs including the total run time and the parameter $\gamma \in (0, 1]$, which is basically the ratio of the total run time that should be assigned to Stage 1 of the Algorithm. Consequently, $(1-\gamma)$ is the ratio of the total run time that should be assigned to Stage 2 of the Algorithm. Since the second stage of the algorithm is only designed to further improve the results of the first stage, we have computationally observed that γ is better to be set to $\frac{2}{3}$. Algorithm 1 shows a precise description of the proposed method. We now make one final comment:

- Algorithms proposed in Stages 1 and 2 are similar and they are both feasibility pump based heuristics. However, in Stage 1, the focus is more on the diversity of the points in the approximate nondominated frontier but, in Sage 2, the focus is more on the quantity of the points in the approximate nondominated frontier. Consequently, the main technical difference between these two stages is that, in Stage 1, the search for finding locally efficient solutions will continue in the not yet dominated regions of the criterion space in each iteration. In order to do so, the set of upper bound points will be updated frequently during the course of Stage 1. However, in the second stage, this condition is relaxed with the hope of finding more locally efficient solutions. Moreover, a variation of the local branching heuristic (see for instance [21, 36]) is incorporated in the second stage for the same purpose.

3.1 Stage 1

The heuristic algorithm (of Stage 1) is specifically designed for finding locally nondominated points from different parts of the criterion space. In other words, the focus of this stage is on the diversity of the points in the approximate nondominated frontier. The heuristic algorithm in this stage is initialized by $\tilde{\mathcal{X}}$, the $Tabu$ list, and the time limit for stage 1. The algorithm maintains a queue of upper bound points, denoted by Q. The queue will be initialized by the point $+\boldsymbol{\infty} = (+\infty, \ldots, +\infty)$. The algorithm terminates when the run time violates the imposed time limit or when the queue is empty. Next we explain the workings of the heuristic algorithm (of Stage 1) at any arbitrary iteration.

In each iteration, the algorithm pops out an element, i.e., an upper bound point, from the queue, which is denoted by $\boldsymbol{u}$. Note that when an element is popped out then that element does



(a) The search regioin defined by $\boldsymbol{u}$ (b) The outcome of the feasibility pump operation (c) The outcome of the local search operation

Figure 3: An illustration of the key operations of Stage 1 of the algorithm when $p = 2$

not exist in the queue anymore. Next, an operation called the weighted sum method, denoted by $WightedSumMethod(\boldsymbol{u})$, is applied to the search region defined by $\boldsymbol{u}$, i.e., $\boldsymbol{u} - \mathbb{R}^p_>$. An illustration of the search region defined by $\boldsymbol{u}$ when $p = 2$ is shown in Figure 3a. This operation is explained in detail in Section 4. This operation simply reports a set of possibly *fractional* solutions, denoted by $\mathcal{X}^f$, that their images in the criterion space are in the search region defined by $\boldsymbol{u}$. Note that we use the term 'fractional solution', when a solution is not feasible for its corresponding MOMILP but it is feasible for its LP-relaxation.

Next, the feasibility pump operation, denoted by FeasibilityPump($\mathcal{X}^f, \boldsymbol{u}, Tabu$), is applied to $\mathcal{X}^f$ for finding a set of (integer) feasible solutions, denoted by $\mathcal{X}^I$, in the search region defined by $\boldsymbol{u}$. This operation is explained in detail in Section 5. An illustration of the set of (integer) feasible solutions in the criterion space produced by the feasibility pump operation (when $p = 2$) is shown in Figure 3b.

If the feasibility pump operation succeeds, i.e., $\mathcal{X}^I \neq \emptyset$, then we try to further improve $\mathcal{X}^I$ both in terms of the quality and quantity using a local search technique which is explained in detail in Section 6. We denote the outcome of this operation again by $\mathcal{X}^I$. The algorithm then updates $\tilde{\mathcal{X}}$ by adding new elements of $\mathcal{X}^I$ to $\tilde{\mathcal{X}}$. An illustration of the set of (integer) feasible solutions in the criterion space (when $p = 2$) produced by the local search operation is shown in Figure 3c.

To avoid spending too much of the valuable computational time on updating the set of upper bound points in the queue Q (in each iteration), we try to update the queue only if it becomes empty. In that case, we find new upper bound points based on the new feasible solutions that have been discovered since the last time that the queue has been updated. In order to do so, the algorithm maintains a copy of $\tilde{\mathcal{X}}$, denoted by $\mathcal{X}^{last}$, of the last time that the queue has been updated.

Algorithm 2 shows a precise description of the proposed heuristic for Stage 1. We now make one final comment:

- The proposed algorithm in this stage can naturally employ parallelism. In other words, each element of the queue Q can be explored in a different processor. We show the value of this feature in Section 7.

Algorithm 2: *Stage*1_*Algorithm*$\big(\tilde{\mathcal{X}}, Tabu, Time\_Limit\_Stage1\big)$

Queue.create(Q);
Queue.add$\big(Q, +\boldsymbol{\infty}\big)$
$\mathcal{X}^{last} \leftarrow \emptyset$
while *time* $\leq$ *Time_Limit_Stage1 & not Queue.empty*(Q) **do**
    Queue.pop$(Q, \boldsymbol{u})$
    $\mathcal{X}^f \leftarrow$ WeightedSumMethod$(\boldsymbol{u})$
    $(\mathcal{X}^I, Tabu) \leftarrow$ FeasibilityPump$(\mathcal{X}^f, \boldsymbol{u}, Tabu)$
    if $\mathcal{X}^I \neq \emptyset$ **then**
        $\mathcal{X}^I \leftarrow$ LocalSearch$(\mathcal{X}^I, \boldsymbol{u})$
        $\tilde{\mathcal{X}} \leftarrow \tilde{\mathcal{X}} \cup \mathcal{X}^I$
    if *Queue.empty*(Q) **then**
        $\mathcal{U} \leftarrow$ Decompose$(\tilde{\mathcal{X}} \backslash \mathcal{X}^{last})$
        foreach $\boldsymbol{u}' \in \mathcal{U}$ **do**
            Queue.add$\big(Q, \boldsymbol{u}'\big)$
        $\mathcal{X}^{last} \leftarrow \tilde{\mathcal{X}}$
return $\big($Pareto$(\tilde{\mathcal{X}}), Tabu\big)$

3.2 Stage 2

Two key differences of this stage with the previous one are that:

- Instead of using the weighted sum method for generating a set of fractional solutions, a variation of the local branching approach (see for instance [21, 36]) is used.
- The upper bound points are not computed in this stage and so the search is not limited to the regions defined by these points.

Overall, these two key differences help us find more feasible solutions. So, the focus of Stage 2 is on the quantity of the points in the approximate nondominated frontier. Before presenting the details of the algorithm, we explain the local branching operation which is denoted by

$$\text{LocalBranching}(S, UpperLimit, Step),$$

where $\mathcal{X}^{new} \subseteq \mathcal{X}$ and $UpperLimit, Step \in \mathbb{Z}_{\geq}$ are its inputs. The output of this operation is a set of possibly fractional solutions denoted by $\mathcal{X}^f$. In this operation, for each $(\boldsymbol{x}'_1, \boldsymbol{x}'_2) \in \mathcal{X}^{new}$, the following optimization problem should be solved:

$$(\boldsymbol{x}^*_1, \boldsymbol{x}^*_2) = \arg\min \Big\{ \sum_{i=1}^{p} z_i(\boldsymbol{x}_1, \boldsymbol{x}_2) : \ (\boldsymbol{x}_1, \boldsymbol{x}_2) \in LP(\mathcal{X}),$$
$$UpperLimit - Step \leq \sum_{j=1:\ x'_{1,j}=0}^{n_1} x_{1,j} + \sum_{j=1:\ x'_{1,j}=1}^{n_1} (1 - x_{1,j}) \leq UpperLimit \Big\}.$$

It is evident that this optimization problem seeks to produce a (possibly fractional) solution in the LP-relaxation of the problem such that the difference between its values of the integer decision variables from those of solution $(\boldsymbol{x}'_1, \boldsymbol{x}'_2)$ to be within a specific neighborhood. The objective function of the optimization problem ensures that the new (possibly fractional) solution to be close to the exact nondominated frontier of the corresponding MOMILP. If $(\boldsymbol{x}^*_1, \boldsymbol{x}^*_2)$ exists and it is not already in $\mathcal{X}^f$ then we add it to $\mathcal{X}^f$.

We now explain the heuristic algorithm (of Stage 2) in detail. The algorithm is initialized by $\tilde{\mathcal{X}}$, the *Tabu* list, and the time limit for Stage 2. The initial values of *UpperBound* and *Step* are set to 2 and 1, respectively. It is worth mentioning that we could initialize $UpperBound = 1$ and $Step = 0$, but this case will be naturally explored in the feasibility pump operation (and so we have tried to remove any redundant calculation). The algorithm terminates when the run time violates the imposed time limit or $UpperLimit > n_1$. In each iteration, the algorithm makes a copy of $\tilde{\mathcal{X}}$ and denote it by $\mathcal{X}^{new}$. Also, the algorithm measures the cardinality of $\tilde{\mathcal{X}}$ at the beginning of each iteration and denote it by *InitialCardinality*. It then explores and updates the set $\mathcal{X}^{new}$ by using a set of operations until it becomes empty. Afterwards, it updates the values of *Step* and *UpperLimit* (based on *InitialCardinality*) before starting the next iteration.

Algorithm 3: $Stage2\_Algorithm(\tilde{\mathcal{X}}, Tabu, Time\_Limit\_Stage2)$

```
UpperLimit ← 2
Step ← 1
while UpperLimit ≤ n_1 & time ≤ Time_Limit_Stage2 do
    X^new ← X̃
    InitialCardinality ← |X̃|
    SearchDone ← False
    while SearchDone = False do
        X^f ← LocalBranching(X^new, UpperLimit, Step)
        (X^I, Tabu) ← FeasibilityPump(X^f, +∞, Tabu)
        if X^I ≠ ∅ then
            X^I ← LocalSearch(X^I, +∞)
            X^new ← NewSolutions(X^I, X̃)
            X̃ ← X̃ ∪ X^I
        else
            X^new ← ∅
        if X^new = ∅ then
            SearchDone ← True
    if |X̃| − InitialCardinality ≤ UpperLimit then
        Step ← Step + 1
    UpperLimit ← UpperLimit + Step + 1
return Pareto(X̃)
```

In particular, to explore and update the set $\mathcal{X}^{new}$ in each iteration, the following steps are conducted. The algorithm first calls LocalBranching$(\mathcal{X}^{new}, UpperLimit, Step)$ to compute a set of possibly fractional solutions $\mathcal{X}^f$. Next, it calls FeasibilityPump$(\mathcal{X}^f, +\infty, Tabu)$ for finding a set

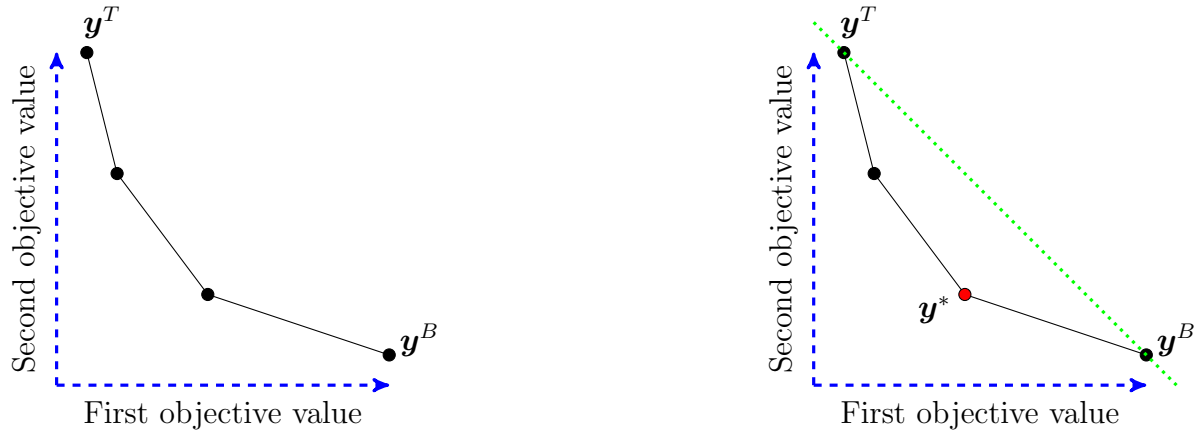
of (integer) feasible solutions, i.e., $\mathcal{X}^I$. If the feasibility pump operation fails, i.e., $\mathcal{X}^I = \emptyset$, then the algorithm sets $\mathcal{X}^{new} = \emptyset$ since it was not able to find new feasible solutions. Otherwise, i.e., $\mathcal{X}^I \neq \emptyset$, the algorithm attempts to further improve $\mathcal{X}^I$ both in terms of the quality and quantity by calling the local search operation. We denote the outcome of this operation again by $\mathcal{X}^I$. The algorithm then sets $\mathcal{X}^{new}$ to the solutions of $\mathcal{X}^I$ in which their corresponding images in the criterion space are different from those in the set $\tilde{\mathcal{X}}$. This operation is denoted by NewSolutions$(\mathcal{X}^I, \tilde{\mathcal{X}})$. The algorithm then updates $\tilde{\mathcal{X}}$ by adding new elements of $\mathcal{X}^I$ to $\tilde{\mathcal{X}}$.

If $\mathcal{X}^{new} \neq \emptyset$ then all the steps (mentioned above) will be repeated for the updated $\mathcal{X}^{new}$. Otherwise, the values of *Step* and *UpperLimit* should be increased. In order to do so, we have employed a simple technique that ensures that if the the number of the new solutions produced in each iteration is not large enough then the value of *UpperLimit* increases with a higher speed. In order to do so, we first check whether

$$|\tilde{\mathcal{X}}| - InitialCardinality \leq UpperLimit.$$

If this is the case then we increase the *Step* by one. Finally, we increase the size of *UpperLimit* by $Step+1$. Algorithm 3 shows a precise description of the proposed heuristic for Stage 2. We now make one final comment:

- The proposed algorithm in this phase can also employ parallelism. In our implementation, if we have p different processors, we simply split $\mathcal{X}^{new}$ into p subsets with (almost) equal size. We then assign each subset to a different processor and apply the local branching, feasibility pump and local search operations on that subset. We show the value of this feature in Section 7.



(a) The endpoints of the nondominated frontier

(b) The imaginary line and the new point

Figure 4: An illustration of the nondominated frontier of a bi-objective linear program

4 The weighted sum method operation

This operation takes an upper bound point $\boldsymbol{u}$ as the input. The operation is basically developed for computing a set of possibly fractional feasible solutions, i.e., $\mathcal{X}^f$, for the LP-relaxation of the

problem that their images in the criterion space lie within the search region defined dy $\boldsymbol{u}$, i.e., $\boldsymbol{u}-\mathbb{R}^p_>$. Specifically, the operation generates and return some of the efficient solutions of the following problem:

$$\min \ \big\{z_1(\boldsymbol{x}_1,\boldsymbol{x}_2),\ldots,z_p(\boldsymbol{x}_1,\boldsymbol{x}_2):\ (\boldsymbol{x}_1,\boldsymbol{x}_2)\in LP(\mathcal{X}),\ z_i(\boldsymbol{x}_1,\boldsymbol{x}_2)<u_i \text{ for all } i\in\{1,\ldots,p\}\big\}. \quad (2)$$

It is worth mentioning that Problem (2) is basically a multi-objective linear program and it is well-known that the nondominated frontier of such a problem is connected. An illustration of the nondominated frontier of a multi-objective linear program with $p=2$ can be found in Figure 4a. Overall, the main goal of the weighted sum method operation is that the generated efficient solutions (mainly) correspond to the *extreme* nondominated points of Problem (2), i.e., circles in Figure 4a. Based on the theory of multi-objective linear programming (see for instance [18]), for any efficient solution $(\boldsymbol{x}_1^*,\boldsymbol{x}_2^*)$ of Problem (2), there exists a weight vector $\boldsymbol{\lambda}\in\mathbb{R}^p_>$ such that:

$$(\boldsymbol{x}_1^*,\boldsymbol{x}_2^*)\in \arg\min\ \big\{\sum_{i=1}^{p}\lambda_i z_i(\boldsymbol{x}_1,\boldsymbol{x}_2):\ (\boldsymbol{x}_1,\boldsymbol{x}_2)\in LP(\mathcal{X}),\ z_i(\boldsymbol{x}_1,\boldsymbol{x}_2)<u_i \text{ for all } i\in\{1,\ldots,p\}\big\}. \quad (3)$$

We denote this operation by ArgWeightedMin$(\boldsymbol{\lambda},\boldsymbol{u})$. So, in order to generate $\mathcal{X}^f$, we only need to update $\boldsymbol{\lambda}$ and solve Problem (3) in each iteration. It is worth mentioning that we have computationally observed that the performance of FPBH improves, if we terminate the weighted sum method operation as soon as the cardinality of $\mathcal{X}^f$ does not satisfy the following condition:

$$|\mathcal{X}^f|\leq CardinalityBound := \lceil\min\{\frac{10}{p}\lceil\log_p h\rceil,\frac{100}{p}\}\rceil,$$

where $h:=\max\{m,n_1+n_2\}$. Note that based on our discussion in Section 2, m is the number of constraints and n_1+n_2 is the number of variables. So, this observation is taken into account in FPBH.

4.1 Instances with two objectives

It is well-known (see for instance [2]) that when $p=2$, updating $\boldsymbol{\lambda}$ can be done efficiently if the top and bottom endpoints of the nondominated frontier are known. So, when $p=2$, we first find the endpoints of the nondominated frontier by using a *lexicographical* technique. For example, the top endpoint, denoted by $\boldsymbol{y}^T:=\boldsymbol{z}(\boldsymbol{x}_1^T,\boldsymbol{x}_2^T)$, can be computed by first solving,

$$(\tilde{\boldsymbol{x}}_1,\tilde{\boldsymbol{x}}_2)\in \arg\min\ \big\{z_1(\boldsymbol{x}_1,\boldsymbol{x}_2):\ (\boldsymbol{x}_1,\boldsymbol{x}_2)\in LP(\mathcal{X}),\ z_i(\boldsymbol{x}_1,\boldsymbol{x}_2)<u_i \text{ for all } i\in\{1,2\}\big\},$$

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

$$(\boldsymbol{x}_1^T,\boldsymbol{x}_2^T)\in \arg\min\ \big\{z_2(\boldsymbol{x}_1,\boldsymbol{x}_2): z_1(\boldsymbol{x}_1,\boldsymbol{x}_2)\leq z_1(\tilde{\boldsymbol{x}}_1,\tilde{\boldsymbol{x}}_2),\ (\boldsymbol{x}_1,\boldsymbol{x}_2)\in LP(\mathcal{X}),\ z_i(\boldsymbol{x}_1,\boldsymbol{x}_2)<u_i \text{ for all } i\in\{1,2\}\big\}.$$

We denote this operation by ArgLexMin$(1,2,\boldsymbol{u})$. By computing $(\boldsymbol{x}_1^T,\boldsymbol{x}_2^T)$, the first potential fractional solution has been generated and so it should be added to $\mathcal{X}^f$. The bottom endpoint, denoted by $\boldsymbol{y}^B:=\boldsymbol{z}(\boldsymbol{x}_1^B,\boldsymbol{x}_2^B)$, can be obtained similarly and so $(\boldsymbol{x}_1^B,\boldsymbol{x}_2^B)$ should also be added to $\mathcal{X}^f$ (if $\boldsymbol{y}^T\neq\boldsymbol{y}^B$). We denote the operation of finding a solution corresponding to the bottom endpoint by ArgLexMin$(2,1,\boldsymbol{u})$. These two points form the first pair of points $(\boldsymbol{y}^T,\boldsymbol{y}^B)$ and they will be added to a queue of pairs of points (if $\boldsymbol{y}^T\neq\boldsymbol{y}^B$ and $|\mathcal{X}^f|\leq CardinalityBound$).

Now, in each iteration, one of the pairs of points in the queue, denoted by $(\boldsymbol{y}^1, \boldsymbol{y}^2)$ where $y_1^1 < y_1^2$ and $y_2^1 > y_2^2$, should be popped out. If there is no pair in the queue or $|\mathcal{X}^f| > CardinalityBound$ then the search terminates and the set $\mathcal{X}^f$ should be reported. Otherwise, we set $\lambda_1 = y_2^1 - y_2^2$ and $\lambda_2 = y_1^2 - y_1^1$ and we call ArgWeightedMin$(\boldsymbol{\lambda}, \boldsymbol{u})$ to compute a new (possibly fractional) efficient solution of Problem (2), denoted by $(\boldsymbol{x}_1^*, \boldsymbol{x}_2^*)$. Note that the generated weight vector results in an objective function which is parallel to the imaginary line that connects $\boldsymbol{y}^1$ and $\boldsymbol{y}^2$ in the criterion space (see Figure 4b). Now, let $\boldsymbol{y}^* := \boldsymbol{z}(\boldsymbol{x}_1^*, \boldsymbol{x}_2^*)$. It is easy to see that $\boldsymbol{y}^*$ is an as yet unknown nondominated point of Problem (2) if $\lambda_1 y_1^* + \lambda_2 y_2^* < \lambda_1 y_1^1 + \lambda_2 y_2^1$. So, in this case, we add $(\boldsymbol{x}_1^*, \boldsymbol{x}_2^*)$ to $\mathcal{X}^f$ and also add two new pairs, i.e., $(\boldsymbol{y}^1, \boldsymbol{y}^*)$ and $(\boldsymbol{y}^*, \boldsymbol{y}^2)$, to the queue. Algorithm 4 shows a precise description of the wighted sum method operation when $p = 2$.

Algorithm 4: WeightedSumMethod$(\boldsymbol{u})$ when $p = 2$

```
Queue.create(Q̃);
X^f ← ∅
(x_1^T, x_2^T) ← ArgLexMin(1, 2, u)
if (x_1^T, x_2^T) ≠ Null then
    y^T ← z(x_1^T, x_2^T)
    X^f ← X^f ∪ {(x_1^T, x_2^T)}
    (x_1^B, x_2^B) ← ArgLexMin(2, 1, u)
    y^B ← z(x_1^B, x_2^B)
    if y^T ≠ y^B & |X^f| ≤ CardinalityBound then
        X^f ← X^f ∪ {(x_1^B, x_2^B)}
        Q̃.add(y^T, y^B)
while |X^f| ≤ CardinalityBound & not Queue.empty(Q̃) do
    Queue.pop(Q̃, (y^1, y^2))
    λ_1 ← y_2^1 − y_2^2
    λ_2 ← y_1^2 − y_1^1
    (x_1^*, x_2^*) ← ArgWeightedMin(λ, u)
    y^* ← z(x_1^*, x_2^*)
    if λ_1 y_1^* + λ_2 y_2^* < λ_1 y_1^1 + λ_2 y_2^1 then
        X^f ← X^f ∪ {(x_1^*, x_2^*)}
        Q̃.add(y^1, y^*)
        Q̃.add(y^*, y^2)
return X^f
```

4.2 Instances with more than two objectives

For MOMILPs with $p > 2$, there exist approaches for updating $\boldsymbol{\lambda}$ (see for instance [30, 35]). However, to avoid spending too much of valuable computational time (on this operation), we apply a heuristic approach instead. In this approach, we first generate p (possibly fractional) efficient solutions of Problem (2). To generate the i-th efficient solution where $i \in \{1, \dots, p\}$, the algorithm first solves,

$$(\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2) \in \arg\min \left\{ z_i(\boldsymbol{x}_1, \boldsymbol{x}_2) : \ (\boldsymbol{x}_1, \boldsymbol{x}_2) \in LP(\mathcal{X}),\ z_{i'}(\boldsymbol{x}_1, \boldsymbol{x}_2) < u_{i'} \text{ for all } i' \in \{1, \dots, p\} \right\},$$

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

$$(\boldsymbol{x}_1^*, \boldsymbol{x}_2^*) \in \arg\min \Big\{ \sum_{i'=1}^{p} z_{i'}(\boldsymbol{x}_1, \boldsymbol{x}_2) : z_i(\boldsymbol{x}_1, \boldsymbol{x}_2) \leq z_i(\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2), \\ (\boldsymbol{x}_1, \boldsymbol{x}_2) \in LP(\mathcal{X}),\ z_{i'}(\boldsymbol{x}_1, \boldsymbol{x}_2) < u_{i'} \text{ for all } i' \in \{1, \ldots, p\} \Big\}.$$

We denote this operation by TwoPhase$(i, \boldsymbol{u})$ where $i \in \{1, \ldots, p\}$ and its corresponding efficient solution $(\boldsymbol{x}_1^*, \boldsymbol{x}_2^*)$ should be added to $\mathcal{X}^f$ [23]. The remaining necessary efficient solutions are generated randomly by creating a set of weight vectors denoted by Λ. Components of each weight vector $\boldsymbol{\lambda} \in \Lambda$ are generated randomly from the standard normal distribution, denoted by RandNormal$(0, 1)$, but we use the absolute value of the generated random numbers. We also normalize the components of each weight vector to assure that the summation of its components is equal to one.

Algorithm 5: WeightedSumMethod$(\boldsymbol{u})$ when $p > 2$

```
1  X^f ← ∅
2  foreach i ∈ {1, ..., p} do
3      if CardinalityBound > 0 then
4          (x*_1, x*_2) ← TwoPhase(i, u)
5          X^f ← X^f ∪ {(x*_1, x*_2)}
6          CardinalityBound ← CardinalityBound − 1
7  Λ ← ∅
8  while |Λ| ≤ CardinalityBound do
9      foreach i ∈ {1, ..., p} do
10         λ_i ← |RandNormal(0, 1)|
11     foreach i ∈ {1, ..., p} do
12         λ_i ← λ_i / (Σ_{j=1}^{p} λ_j)
13     Λ ← Λ ∪ {λ}
14 foreach λ ∈ Λ do
15     (x*_1, x*_2) ← ArgWeightedMin(λ, u)
16     X^f ← X^f ∪ {(x*_1, x*_2)}
17 X^f ← Distinct(X^f)
18 return X^f
```

Next, for each $\boldsymbol{\lambda} \in \Lambda$, we call ArgWeightedMin$(\boldsymbol{\lambda}, \boldsymbol{u})$ to compute a new (possibly fractional) efficient solution for Problem (2), denoted by $(\boldsymbol{x}_1^*, \boldsymbol{x}_2^*)$, and add it to $\mathcal{X}^f$. Algorithm 5 shows a precise description of the weighted sum method operation when $p > 2$. During the course of the algorithm, the number of efficient solutions generated by the algorithm is no more than *CardinalityBound*. Moreover, the algorithm removes equivalent solutions, meaning if there are several efficient solutions with the same image in the criterion space, only one of them will be reported. This operation is denoted by Distinct$(\mathcal{X}^f)$ in Algorithm 5.

5 The feasibility pump operation

Our proposed feasibility pump approach is similar to the feasibility pump approach used for single-objective integer linear programs. The single-objective version of the feasibility pump works on a pair of solutions $(\boldsymbol{x}_1^f, \boldsymbol{x}_2^f)$ and $(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I)$. The first one is feasible for the LP-relaxation of the problem, and so it may be fractional. However, the latter is integer but it is not necessarily feasible for Problem (1), i.e., geometrically it may lie outside the polyhedron corresponding to the LP-relaxation of the problem. The feasibility pump iteratively updates $(\boldsymbol{x}_1^f, \boldsymbol{x}_2^f)$ and $(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I)$, and in each iteration, the hope is to further reduce the distance between them. Consequently, ultimately, we can hope to obtain an (integer) feasible solution.

In light of the above, we now present our proposed feasibility pump operation. This operation takes a set of possibly fractional solutions $\mathcal{X}^f$, an upper bound point $\boldsymbol{u}$, and the *Tabu* list as inputs, and returns the updated *Tabu* list, and a set of (integer) feasible solutions, $\mathcal{X}^I$. At the beginning, we set $\mathcal{X}^I = \emptyset$. Also, at the end, we call DISTINCT($\mathcal{X}^I$) to remove the equivalent solutions of $\mathcal{X}^I$ before returning it.

The feasibility pump operation attempts to construct some integer feasible solutions in the search region defined by $\boldsymbol{u}$, i.e., $\boldsymbol{u} - \mathbb{R}^p_>$, based on each $(\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2) \in \mathcal{X}^f$. By construction of FPBH, if $\tilde{\boldsymbol{x}}_1 \in \{0,1\}^{n_1}$ then we must have that $(\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2) \in \mathcal{X}$, $z_i(\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2) < u_i$ for all $i = 1, \ldots, p$. So, in that case, $(\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2)$ is an integer feasible solution in the search region defined by $\boldsymbol{u}$, and so $(\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2)$ should be added to $\mathcal{X}^I$.

Let e be the base of the natural logarithm. If $\tilde{\boldsymbol{x}}_1 \notin \{0,1\}^{n_1}$ then the algorithm attempts to construct at most $|\Theta| = p + 1$ integer feasible solutions based on $(\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2)$ where,

$$\Theta := \{0, \frac{e^1}{\sum_{j=1}^p e^j}, \ldots, \frac{e^p}{\sum_{j=1}^p e^j}\},$$

is a set of weights to modify the *distance* function in the feasibility pump operation (at the end of this section, the details of the distance function are explained). Specifically, for each $\theta \in \Theta$, we set the maximum number of attempts for constructing an integer feasible solution based on $(\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2)$ to be equal to the number of variables with fractional values in $\tilde{\boldsymbol{x}}_1$, denoted by FRACTIONALDEGREE($\tilde{\boldsymbol{x}}_1$). For a given $(\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2)$ and θ, before starting the set of attempts to generate an integer feasible solution, we make a copy of $(\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2)$ and denote it by $(\boldsymbol{x}_1^f, \boldsymbol{x}_2^f)$. The set of attempts will be conducted on $(\boldsymbol{x}_1^f, \boldsymbol{x}_2^f)$. As soon as an integer feasible solution is found, the algorithm will terminate its search for constructing integer feasible solutions based on $(\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2)$ and θ. Next, we explain the workings of the algorithm during each attempt for constructing an integer feasible solution based on $(\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2)$ and θ. Note that in the remaining, whenever we say that 'the search will terminate', we mean that the search for constructing an integer feasible solution based on $(\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2)$ and θ will terminate.

During each attempt, the algorithm first rounds each component of $\boldsymbol{x}_1^f$. This operation is denoted by ROUND($\boldsymbol{x}_1^f$). We define $(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I) := (\text{ROUND}(\tilde{x}_1), \boldsymbol{x}_2^f)$. Next, the algorithm checks whether $(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I) \in \mathcal{X}$ and $z_i(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I) < u_i$ for all $i \in \{1, \ldots, p\}$. If that is the case then the algorithm has found an integer feasible solution in the search region defined by $\boldsymbol{u}$. So, the search will terminate after adding $(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I)$ to $\mathcal{X}^I$. So, in the remaining we assume that the search does not terminate after computing $(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I)$.

In this case, the algorithm first checks whether $\boldsymbol{x}_1^I$ is in the *Tabu* list. If it is then the algorithm tries to modify $\boldsymbol{x}_1^I$ by using a flipping operation that flips some components of $\boldsymbol{x}_1^I$, i.e., if a component has

Algorithm 6: FeasibilityPump($\mathcal{X}^f, \boldsymbol{u}, Tabu$)

$\mathcal{X}^I \leftarrow \emptyset$
foreach $(\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2) \in \mathcal{X}^f$ **do**
  if $\tilde{\boldsymbol{x}}_1 \in \{0,1\}^{n_1}$ **then**
    $\mathcal{X}^I \leftarrow \mathcal{X}^I \cup \{(\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2)\}$
  else
    $Max\_Iteration \leftarrow$ FractionalDegree($\tilde{\boldsymbol{x}}_1$)
    foreach $\theta \in \Theta$ **do**
      $SearchDone \leftarrow False$
      $t \leftarrow 1$
      $(\boldsymbol{x}_1^f, \boldsymbol{x}_2^f) \leftarrow (\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2)$
      while $SearchDone = False$ & $t \leq Max\_Iteration$ **do**
        $(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I) \leftarrow \big($Round$(\boldsymbol{x}_1^f), \boldsymbol{x}_2^f\big)$
        if $(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I) \in \mathcal{X}$ & $z_1(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I) < u_1$ & $\ldots$ & $z_p(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I) < u_p$ **then**
          $\mathcal{X}^I \leftarrow \mathcal{X}^I \cup \{(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I)\}$
          $SearchDone \leftarrow True$
        else
          if $\boldsymbol{x}_1^I \in Tabu$ **then**
            $\boldsymbol{x}_1^I \leftarrow$ FlippingOperation($\boldsymbol{x}_1^f, \boldsymbol{x}_1^I, Tabu$)
            if $\boldsymbol{x}_1^I = Null$ **then**
              $SearchDone \leftarrow True$
            else
              if $(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I) \in \mathcal{X}$ & $z_1(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I) < u_1$ & $\ldots$ & $z_p(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I) < u_p$ **then**
                $\mathcal{X}^I \leftarrow \mathcal{X}^I \cup \{(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I)\}$
                $SearchDone \leftarrow True$
          if $SearchDone = False$ **then**
            $Tabu.add(\boldsymbol{x}_1^I)$
            $(\boldsymbol{x}_1^f, \boldsymbol{x}_2^f) \leftarrow$ FeasibilitySearch($\boldsymbol{x}_1^I, \boldsymbol{u}, \theta$)
        $t \leftarrow t + 1$
$\mathcal{X}^I \leftarrow$ Distinct($\mathcal{X}^I$)
return ($\mathcal{X}^I$, $Tabu$)

the value of zero then we may change it to one, and the other way around. We explain the details of this operation in Section 5.1. We denote the flipping operation by FLIPPINGOPERATION($Tabu, \boldsymbol{x}_1^f, \boldsymbol{x}_1^I$), and it returns an updated $\boldsymbol{x}_1^I$. If $\boldsymbol{x}_1^I = Null$ then the flipping operation has failed and in this case, the algorithm will terminate its search. However, if $\boldsymbol{x}_1^I \neq Null$ then there is a possibility that $(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I)$ to be an integer feasible solution. So, we again check whether $(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I) \in \mathcal{X}$ and $z_i(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I) < u_i$ for all $i \in \{1, \ldots, p\}$. If that is the case then we have found an integer feasible solution. So, again the search will terminate after adding $(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I)$ to $\mathcal{X}^I$.

Finally, regardless of whether $\boldsymbol{x}_1^I$ is in the $Tabu$ list or not, if the search has not terminated yet, the algorithm takes two further steps:

- It adds $\boldsymbol{x}_1^I$ to the $Tabu$ list; and
- It updates $(\boldsymbol{x}_1^f, \boldsymbol{x}_2^f)$ using $\boldsymbol{x}_1^I$ by solving the following optimization problem,

$$
\begin{aligned}
(\boldsymbol{x}_1^f, \boldsymbol{x}_2^f) = \arg\min \; \Big\{ & (1-\theta)\Delta(\boldsymbol{x}_1^I) + \theta \frac{\sum_{i=1}^{p} z_i(\boldsymbol{x}_1, \boldsymbol{x}_2)}{\sqrt{\sum_{j=1}^{n_1} (\sum_{i=1}^{p} c_{ij})^2 + \sum_{j=1}^{n_2} (\sum_{i=1}^{p} d_{ij})^2}} : \\
& (\boldsymbol{x}_1, \boldsymbol{x}_2) \in LP(\mathcal{X}), \\
& z_i(\boldsymbol{x}_1, \boldsymbol{x}_2) < u_i \qquad \text{for all } i \in \{1, \ldots, p\} \Big\},
\end{aligned}
$$

where

$$
\Delta(\boldsymbol{x}_1^I) := \sum_{j=1:\; x_{1,j}^I = 0}^{n_1} x_{1,j} + \sum_{j=1:\; x_{1,j}^I = 1}^{n_1} (1 - x_{1,j}).
$$

The goal of this optimization problem is to find a (possibly fractional) solution, which is closer to $(\boldsymbol{x}_1^I, \boldsymbol{x}_2^I)$. We denote this optimization problem by FEASIBILITYSEARCH($\boldsymbol{x}_1^I \boldsymbol{u}, \theta$), and its objective function is the distance function that we have used in our study. The first part of the distance function tries to minimize the distance between the new $\boldsymbol{x}_1^f$ and $\boldsymbol{x}_1^I$, but the second part is only for modifying the distance function based on the normalized value of $\sum_{i=1}^{p} z_i(\boldsymbol{x}_1, \boldsymbol{x}_2)$. Note that the distance function that we have used is similar to the typical distance function used in the feasibility pump heuristic for single-objective optimization problems [20, 22]. Only the second part of the distance function is modified here since we are dealing with multiple objectives. Note too that the set Θ introduced at the beginning of this section performs the best for our test instances. In practice, we realized that the existence of a linear relationship between elements of Θ does not help. So, we generated Θ such that there is a non-linear relationship between its elements.

Algorithm 6 shows a precise description of the proposed feasibility pump operation. We now make one final comment:

- In practice, we have observed that if it turns out that $\boldsymbol{x}_1^I = \emptyset$ at Line 19 of Algorithm 6 then it is unlikely to find any feasible solution for the remaining elements of Θ. So, in this case, it is better to directly return to Line 2 of the code for selecting a new element of $\mathcal{X}^f$.

5.1 The flipping operation

The flipping operation takes the $Tabu$ list, $\boldsymbol{x}_1^f$ (which is fractional), and the integer solution $\boldsymbol{x}_1^I$ as inputs. Obviously, by construction, $\boldsymbol{x}_1^I$ is in the Tabu list. So, this operation attempts to flip the components of $\boldsymbol{x}_1^I$ with the hope that the outcome to not be an element of the $Tabu$ list. As soon as one is found, the flipping operation terminates, and the updated $\boldsymbol{x}_1^I$ will be returned.

At the beginning of the operation, we construct two vectors of variables denoted by $\boldsymbol{x}_1^*$ and $\widehat{\boldsymbol{x}}_1^I$. Specifically, we set $\boldsymbol{x}_1^* = null$ at the beginning of the operation, but at the end, this vector contains the updated value of $\boldsymbol{x}_1^I$ (if we find any) that should be reported. The vector $\widehat{\boldsymbol{x}}_1^I$ is a copy of (initial value) of $\boldsymbol{x}_1^I$ and all necessary flips will be conducted on $\widehat{\boldsymbol{x}}_1^I$.

Overall, the flipping operation, contains two parts including a deterministic procedure and a stochastic procedure. The stochastic procedure will be activated only if the deterministic fails to update $\boldsymbol{x}_1^I$, i.e., if $\boldsymbol{x}_1^*$ is still $null$ after the deterministic procedure. It is worth mentioning that we have computationally observed that this order performs better than first doing the stochastic procedure and then deterministic procedure.

Before explaining these two procedures, a new operation denoted by $\textsc{SortIndex}(\boldsymbol{x}_1^f, \boldsymbol{x}_1^I)$ is introduced. This operation simply sorts the set $\{j \in \{1, \ldots, n\} : |x_{1,j}^f - x_{1,j}^I| \neq 0\}$ based on the value of $|x_{1,j}^f - x_{1,j}^I|$ from large to small. Note that, by construction, if $x_{1,j}^f$ is not fractional then $x_{1,j}^I = x_{1,j}^f$ (because $x_{1,j}^I$ is the rounded value of $x_{1,j}^f$). This implies that this operation sorts the index set of fractional components of $\boldsymbol{x}_1^f$. We denote the result of this operation by $\{s^1, \ldots, s^M\}$ where

$$|x_{1,s^k}^f - x_{1,s^k}^I| \geq |x_{1,s^{k+1}}^f - x_{1,s^{k+1}}^I|$$

for each $k = 1, \ldots, M-1$.

The deterministic procedure attempts at most M times to update $\boldsymbol{x}_1^I$. At the beginning, we set $\widehat{\boldsymbol{x}}_1^I = \boldsymbol{x}_1^I$. During attempt $j \leq M$, we flip the component $\widehat{x}_{1,s^j}^I$. This implies that if $\widehat{x}_{1,s^j}^I = 1$ then we make it equal to zero, and the other way around. We denote this operation by $\textsc{Flip}(\widehat{\boldsymbol{x}}_1^I, s^j)$. If the result is not in the $Tabu$ list then the algorithm terminates, and we set $\boldsymbol{x}_1^*$ to $\widehat{\boldsymbol{x}}_1^I$. Otherwise, (if possible) a new attempt will be started.

The stochastic procedure also attempts at most M times to construct an integer solution. At each attempt, we first set $\widehat{\boldsymbol{x}}_1^I = \boldsymbol{x}_1^I$ and then we randomly select an integer number from the interval $[\lceil \frac{M}{2} \rceil, M-1]$. This operation is denoted by $\textsc{RandBetween}(\lceil \frac{M}{2} \rceil, M-1)$, and its result is denoted by Num. We then randomly generate a set denoted by R such that $R \subseteq \{1, \ldots, M\}$ and $|R| = Num$. The operation is denoted by $\textsc{RandomlyChoose}(Num, \{1, \ldots, M\})$. We then flip the component $\widehat{x}_{1,s^r}^I$ for each $r \in R$. If the result is not in the $Tabu$ list then the algorithm terminates, and we set $\boldsymbol{x}_1^*$ to $\widehat{\boldsymbol{x}}_1^I$. Otherwise, (if possible) a new attempt will be started.

Algorithm 7 shows a precise description of $\textsc{FlippingOperation}(\boldsymbol{x}_1^f, \boldsymbol{x}_1^I, Tabu)$.

6 The local search operation

The local search operation (that we have developed) has two stages. The operation takes $\mathcal{X}^I$ (which is a subset of the feasible solutions of Problem (1)) and $\boldsymbol{u}$ as inputs, and reports an updated $\mathcal{X}^I$. The first stage will be only activated when $n_1 > 0$ and $n_2 > 0$, i.e., the instance is a mixed integer linear program. The second stage will be immediately called after the first stage regardless of whether the

instance is a pure integer program or mixed integer program. A precise description of the local search operation can be found in Algorithm 8.

Algorithm 7: FLIPPINGOPERATION($\boldsymbol{x}_1^f, \boldsymbol{x}_1^I, Tabu$)

$\{s^1, \ldots, s^M\} \leftarrow$ SORTINDEX($\boldsymbol{x}_1^f, \boldsymbol{x}_1^I$)
$\boldsymbol{x}_1^* \leftarrow Null$
$\widehat{\boldsymbol{x}}_1^I \leftarrow \boldsymbol{x}_1^I$
$j \leftarrow 1$
while $j \leq M$ *and* $\boldsymbol{x}_1^* = Null$ **do**
  $\widehat{\boldsymbol{x}}_1^I \leftarrow$ FLIP($\widehat{\boldsymbol{x}}_1^I, s^j$)
  if $\widehat{\boldsymbol{x}}_1^I \notin Tabu$ **then**
    $\boldsymbol{x}_1^* \leftarrow \widehat{\boldsymbol{x}}_1^I$
  else
    $j \leftarrow j + 1$
$j \leftarrow 1$
while $j \leq M$ *and* $\widehat{\boldsymbol{x}}_1^* = Null$ **do**
  $\widehat{\boldsymbol{x}}_1^I \leftarrow \boldsymbol{x}_1^I$
  $Num \leftarrow$ RANDBETWEEN($\lceil \frac{M}{2} \rceil, M - 1$)
  $R \leftarrow$ RANDOMLYCHOOSE($Num, \{1, \ldots, M\}$)
  foreach $r \in R$ **do**
    $\widehat{\boldsymbol{x}}_1^I \leftarrow$ FLIP($\widehat{\boldsymbol{x}}_1^I, s^r$)
  if $\widehat{\boldsymbol{x}}_1^I \notin Tabu$ **then**
    $\boldsymbol{x}_1^* \leftarrow \widehat{\boldsymbol{x}}_1^I$
  else
    $j \leftarrow j + 1$
$\boldsymbol{x}_1^I \leftarrow \boldsymbol{x}_1^*$
return $\boldsymbol{x}_1^I$

6.1 The local search - Stage 1

The operation developed for Stage 1 takes $\mathcal{X}^I$ (which is a subset of the feasible solutions of Problem (1)) and $\boldsymbol{u}$ as inputs, and reports an updated $\mathcal{X}^I$. The key idea behind this operation comes from the fact that any $(\boldsymbol{x}_1, \boldsymbol{x}_2) \in \mathcal{X}^I$ is an integer feasible solution for Problem (1). So, if we set the integer decision variables of the corresponding MOMILP to $\boldsymbol{x}_1$, a multi-objective linear program will be constructed. Obviously, all feasible solutions of such a multi-objective linear program are also feasible for the corresponding MOMILP. Consequently, we can attempt to enumerate some of the extreme nondominated points of such a multi-objective linear program since they may be part of the true nondominated frontier of the corresponding MOMILP.

In light of the above, in this operation, we first make a copy of $\mathcal{X}^I$ and denote it by $\widehat{\mathcal{X}}$. We start by first removing redundant elements of $\widehat{\mathcal{X}}$. This implies that for any $(\widehat{\boldsymbol{x}}_1, \widehat{\boldsymbol{x}}_2), (\widehat{\boldsymbol{x}}'_1, \widehat{\boldsymbol{x}}'_2) \in \widehat{\mathcal{X}}$ with $\widehat{\boldsymbol{x}}_1 = \widehat{\boldsymbol{x}}'_1$ only one of them should remain in the set. We denote this operation by UNIQUE($\widehat{\mathcal{X}}$) and its outcome

Algorithm 8: LocalSearch($\mathcal{X}^I, \boldsymbol{u}$)

```
if n_1 > 0 and n_2 > 0 then
    X^I ← LocalSearch-Stage1(X^I, u)
X^I ← LocalSearch-Stage2(X^I, u)
return X^I
```

is denoted by $\widehat{\mathcal{X}}$ again.

We also denote the set of new feasible solutions by $\mathcal{X}^*$ which is empty at the beginning of the operation. For each $(\widehat{\boldsymbol{x}}_1, \widehat{\boldsymbol{x}}_2) \in \widehat{\mathcal{X}}$, we call an operation entitled ModifiedWeightedSumMethod($\widehat{\boldsymbol{x}}_1, \boldsymbol{u}$) which is precisely the weighted sum operation that we explained in Section 4 with two small differences. Basically, the first difference is that the following constraints should be added to any model that should be solved in the weighted sum operation:

$$x_{1,j} = \widehat{x}_{1,j} \qquad \forall j \in \{1, \dots, n_1\}.$$

The second difference is that to avoid spending too much of valuable computational time on the modified wighted sum method, we set

$$CardinalityBound = \lceil \frac{\min\{\frac{10}{p}\lceil \log_p h \rceil, \frac{100}{p}\}}{|\,\textsc{Unique}(\mathcal{X}^I)|} \rceil.$$

The set of solutions found by ModifiedWeightedSumMethod($\widehat{\boldsymbol{x}}_1, \boldsymbol{u}$) must be feasible for Problem (1) (and lie within the search region defined by $\boldsymbol{u}$), and so they will be added to $\mathcal{X}^*$. Finally, we call Distinct($\mathcal{X}^I \cup \mathcal{X}^*$) to compute the distinct elements of $\mathcal{X}^I \cup \mathcal{X}^*$ and report them as the updated $\mathcal{X}^I$.

A precise description of Stage 1 of the local search operation can be found in Algorithm 9.

Algorithm 9: LocalSearch-Stage1($\mathcal{X}^I, \boldsymbol{u}$)

```
X̂ ← X^I
X̂ ← Unique(X̂)
X* ← ∅
foreach (x̂_1, x̂_2) ∈ X̂ do
    X* ← X* ∪ ModifiedWeightedSumMethod(x̂_1, u)
X^I ← Distinct(X^I ∪ X*)
return X^I
```

6.2 The local search - Stage 2

The operation developed for Stage 2 also takes $\mathcal{X}^I$ (which is a subset of the feasible solutions of Problem (1)) and $\boldsymbol{u}$ as inputs, and reports an updated $\mathcal{X}^I$ after calling Pareto($\mathcal{X}^I$). In this operation, the algorithm first makes a copy of $\mathcal{X}^I$ and denote it by $\widehat{\mathcal{X}}$. For each $(\widehat{\boldsymbol{x}}_1, \widehat{\boldsymbol{x}}_2) \in \widehat{\mathcal{X}}$, the algorithm tries

to generate at most n_1 new (integer) feasible solutions for Problem (1) that lie within the search region defined by $\boldsymbol{u}$, and add them to $\mathcal{X}^l$. More precisely, for each $(\widehat{\boldsymbol{x}}_1, \widehat{\boldsymbol{x}}_2) \in \widehat{\mathcal{X}}$ and each $j \in \{1, \ldots, n_1\}$, the algorithm first makes a copy of $(\widehat{\boldsymbol{x}}_1, \widehat{\boldsymbol{x}}_2)$, denoted by $(\boldsymbol{x}_1^{new}, \boldsymbol{x}_2^{new})$. It then explores the consequence of flipping component $x_{1,j}^{new}$. Obviously, if $x_{1,j}^{new} = 1$ and $c_{1,j} > 0$ or $c_{2,j} > 0$ or $\ldots$ or $c_{p,j} > 0$ then the flipping can improve the value of at least one objective function. So, in this case, the algorithm checks whether the new solution after flipping is feasible for Problem (1) and lies within the search region defined by $\boldsymbol{u}$, and if that is the case then the algorithm adds it to $\mathcal{X}^I$. Similarly, if $x_{1,j}^{new} = 0$ and either $c_{1,j} < 0$ or $c_{2,j} < 0$ or $\ldots$ or $c_{p,j} < 0$, then the flipping can improve the value of at least one objective function. So, again, in this case, it checks whether the new solution after flipping is feasible for Problem (1) and lies within the search region defined by $\boldsymbol{u}$, and if that is the case then the algorithm adds it to $\mathcal{X}^I$.

Algorithm 10: LocalSearch-Stage2($\mathcal{X}^I, \boldsymbol{u}$)

$\widehat{\mathcal{X}} \leftarrow \mathcal{X}^I$
foreach $(\widehat{\boldsymbol{x}}_1, \widehat{\boldsymbol{x}}_2) \in \widehat{\mathcal{X}}$ **do**
  foreach $j \in \{1, \ldots, n_1\}$ **do**
    $(\boldsymbol{x}_1^{new}, \boldsymbol{x}_2^{new}) \leftarrow (\widehat{\boldsymbol{x}}_1, \widehat{\boldsymbol{x}}_2)$
    if $x_{1,j} = 1$ *and* $(c_{1,j} > 0$ *or* $\ldots$ *or* $c_{p,j} > 0)$ **then**
      $x_{1,j}^{new} \leftarrow 0$
      if $(\boldsymbol{x}_1^{new}, \boldsymbol{x}_2^{new}) \in \mathcal{X}$ *and* $z_1(\boldsymbol{x}_1^{new}, \boldsymbol{x}_2^{new}) < u_1$ *and* $\ldots$ *and* $z_p(\boldsymbol{x}_1^{new}, \boldsymbol{x}_2^{new}) < u_p$ **then**
        $\mathcal{X}^I \leftarrow \mathcal{X}^I \cup \{(\boldsymbol{x}_1^{new}, \boldsymbol{x}_2^{new})\}$
    else
      if $x_{1,j} = 0$ *and* $(c_{1,j} < 0$ *or* $\ldots$ *or* $c_{p,j} < 0)$ **then**
        $x_{1,j}^{new} \leftarrow 1$
        if $(\boldsymbol{x}_1^{new}, \boldsymbol{x}_2^{new}) \in \mathcal{X}$ *and* $z_1(\boldsymbol{x}_1^{new}, \boldsymbol{x}_2^{new}) < u_1$ *and* $\ldots$ *and* $z_p(\boldsymbol{x}_1^{new}, \boldsymbol{x}_2^{new}) < u_p$ **then**
          $\mathcal{X}^I \leftarrow \mathcal{X}^I \cup \{(\boldsymbol{x}_1^{new}, \boldsymbol{x}_2^{new})\}$
$\mathcal{X}^I \leftarrow$ Pareto($\mathcal{X}^I$)
return $\mathcal{X}^I$

Algorithm 10 shows a precise description of LocalSearch-Stage2($\mathcal{X}^I, \boldsymbol{u}$). We note that if there are too many equality constraints in Problem (1), the second stage of the local search operation most likely will fail to generate any new integer feasible solution. So, we only activate this stage if at least 5% of constraints are inequalities.

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 FPBH and the instances are available as open source Julia packages at `https://goo.gl/fwgHjO` and `https://goo.gl/mevqbk`, respectively. The Julia package is compatible with the popular JuMP modeling language (Dunning et al. [16], Lubin and Dunning [27]), supports input in LP and MPS file formats, can plot nondominated frontiers, can compute different quality measures (hypervolume, cardinality, coverage and uniformity), supports execution on multiple processors and can use any linear programming solver supported by MathProgBase.jl (such as CPLEX, Clp, GLPK, etc). An efficient implementation of FPBH specifically using CPLEX.jl for CPLEX is also available as an open source Julia package at `https://goo.gl/Yne4eU`. Note that the default version used in this paper is the efficient implementation of FPBH using CPLEX.jl, since during the course of the algorithm, it does not have to rebuild the linear programming models (used in FPBH) repetitively. All experimental results and the nondominated frontiers of all instances computed in this study are available at `https://goo.gl/xYxtiq` and `https://goo.gl/P4rPrB`, respectively.

In total, 351 instances are used in this study. Some necessary information about these instances such as the size and some studies that have used them can be found in Table 1 where '#Const' and '#Ins' are the number of constraints and instances, respectively. The true nondominated frontier of 291 out of 366 instances are known and have been generously provided to us by the authors listed in Table 1. It is worth mentioning that only 60 out of 351 instances are randomly generated in this study based on the procedure explained in Boland et al. [5]. For these 60 random instances, we have that $p > 2$ and $n_1, n_2 > 0$. Unfortunately, for these instances the true nondominated frontier is unknown since (to the best of our knowledge) there is no exact algorithm that can generate the true nondominated frontier of MOMILP with $p > 2$ and $n_1, n_2 > 0$ in general.

To show the performance of FPBH, we use different versions of the proposed algorithm in this computational study:

- **V1:** This version is designed for showing the importance of the proposed feasibility pump operation. So, the underlying question/idea is that if we want to run just the feasibility pump operation once when $\theta = 0$, then how good would be the solutions obtained by this operation using a single thread (i.e., the parallelization option is off)? So, we define V1 precisely as first calling WeightedSumMethod($+\boldsymbol{\infty}$) to obtain $\mathcal{X}^f$ and then calling FeasibilityPump($\mathcal{X}^f, +\boldsymbol{\infty}, \emptyset$) by imposing $\theta = 0$ to obtain $\mathcal{X}^I$.

- **V2:** This version is designed for showing the importance of the set Θ. So, we define V2 precisely as V1 but imposing $\theta \in \Theta$ to obtain $\mathcal{X}^I$ when using a single thread (i.e., the parallelization option is off);

- **V3:** This version is designed for showing the importance of the local search operation. We define V3 precisely as V2 to obtain $\mathcal{X}^I$ but calling LocalSearch($\mathcal{X}^I, +\boldsymbol{\infty}$) immediately after that to obtain an updated $\mathcal{X}^I$ when using a single thread (i.e., the parallelization option is off).

- **V4:** Note that V3 can be viewed as the first iteration of the Stage 1 of FPBH. So, we define V4 precisely as the entire Stage 1 of FPBH when using a single thread. In other words, we basically impose $\gamma = 1$ in Algorithm 1.

- **V5T1:** We define V5T1 precisely as FPBH, i.e., Algorithm 1 with $\gamma = \frac{2}{3}$, when using a single thread (i.e., the parallelization option is off).

Table 1: The list of instances used in this study

Problem Type	#Obj	#Continuous Var	#Binary Var	#Const	#Ins	Source
Mixed Binary	2	10	10	20	5	Boland et al. [5]
		20	20	40		
		40	40	80		
		80	80	160		
		160	160	320		
Facility Location		800	16	850	4	
		1,250	25	1,300		
		2,500	50	2,550		
Assignment	3	0	25	10	10	Kirlik and Sayın [23]
			100	20		
			225	30		
			400	40		
			625	50		
			900	60		
			1,225	70		
			1,600	80		
			2,025	90		
			2,500	100		
Knapsack	3		10	1		
			20			
			30			
			40			
			50			
			60			
			70			
			80			
			90			
			100			
	4		10			
			20			
			30			
			40		5	
	5		10		10	
			20		9	
Mixed Binary	3	40	40	80	5	**Random** (Boland et al. [5])
		80	80	160		
		160	160	320		
		320	320	640		
	4	40	40	80		
		80	80	160		
		160	160	320		
		320	320	640		
	5	40	40	80		
		80	80	160		
		160	160	320		
		320	320	640		

- **V5T2:** We define V5T2 precisely as FPBH, i.e., Algorithm 1 with $\gamma = \frac{2}{3}$, when using two threads (i.e., the parallelization option is on).
- **V5T3:** We define V5T3 precisely as FPBH, i.e., Algorithm 1 with $\gamma = \frac{2}{3}$, when using three threads (i.e., the parallelization option is on).
- **V5T4:** We define V5T4 precisely as FPBH, i.e., Algorithm 1 with $\gamma = \frac{2}{3}$, when using four threads (i.e., the parallelization option is on).

Note that in this computational study, for almost all, i.e., 291 out of 351, instances, the true nondominated frontier, i.e., $\mathcal{Y}_N$, is known. Also, FPBH and the benchmark algorithm employed in this computational study are all heuristics, and so they just generate an approximate nondominated frontier, denoted by $\mathcal{Y}_N^A$. However, since $\mathcal{Y}_N^A$ is not necessarily equal to $\mathcal{Y}_N$ (in fact $\mathcal{Y}_N^A$ may not be even a subset of $\mathcal{Y}_N$), evaluating the quality of $\mathcal{Y}_N^A$ is crucial [37]. In order to do so, it is common to normalize the points in $\mathcal{Y}_N$ and $\mathcal{Y}_N^A$. We assume that an arbitrary point $\boldsymbol{y} \in \mathbb{R}^p$ in the criterion space should be normalized as follows:

$$\left(\frac{y_1 - \min_{\bar{y}\in\mathcal{Y}_N} \bar{y}_1}{\max_{\bar{y}\in\mathcal{Y}_N} \bar{y}_1 - \min_{\bar{y}\in\mathcal{Y}_N} \bar{y}_1}, \ldots, \frac{y_p - \min_{\bar{y}\in\mathcal{Y}_N} \bar{y}_p}{\max_{\bar{y}\in\mathcal{Y}_N} \bar{y}_p - \min_{\bar{y}\in\mathcal{Y}_N} \bar{y}_p}\right).$$

Obviously this guarantees that if $y \in \mathcal{Y}_N$ then the components of its corresponding normalized point take on values from the interval $[0, 1]$. So, in the remaining, we assume that all points in $\mathcal{Y}_N$ and $\mathcal{Y}_N^A$ are already normalized. We next review a few techniques that can be used to evaluate the quality of an approximate nondominated frontier when the true nondominated frontier is known (for further details please refer to [6]). However, before that, we have to first introduce some new notation. We denote the Euclidean distance between two points $\boldsymbol{y}$ and $\boldsymbol{y}'$ by $d(\boldsymbol{y}, \boldsymbol{y}')$. Define $k(\boldsymbol{y})$ to be the closest point in the approximate frontier to a true nondominated point $\boldsymbol{y} \in \mathcal{Y}_N$. Finally, for each $\boldsymbol{y} \in \mathcal{Y}_N^A$, define $n(\boldsymbol{y})$ to be the number of (true) nondominated points $\boldsymbol{y}' \in \mathcal{Y}_N \setminus \mathcal{Y}_N^A$ with $k(\boldsymbol{y}') = \boldsymbol{y}$.

- **Hypervolume gap.** One of the best-known measures for assessing and comparing approximate frontiers is the *hypervolume indicator* (or S-metric), which is the volume of the dominated part of the criterion space with respect to a reference point [41, 42]. As the reference point impacts the value of the hypervolume indicator, we use the *nadir point*, i.e., $z_i^N := \max\{y_i : \boldsymbol{y} \in \mathcal{Y}_N\}$ for $i = 1, \ldots, p$, as the reference point, which is the best possible choice for the reference point. Note that computing the nadir point is not easy in general, but trivial once the nondominated frontier is known. Given that the entire nondominated frontier is known for all our instances, we define the hypervolume gap as follows,

  $$\frac{100 \times (\text{ Hypervolume of } \mathcal{Y}_N - \text{ Hypervolume of } \mathcal{Y}_N^A)}{\text{Hypervolume of } \mathcal{Y}_N}.$$

  As a consequence, an approximate nondominated frontier with smaller hypervolume gap is more desirable. In this study, we use the package PAGMO (`https://esa.github.io/pagmo2/index.html`) to compute hypervolume [9].

- **Cardinality.** We define the cardinality of an approximate frontier simply as the percentage of the points of the true nondominated points it contains, i.e., $\frac{100|\mathcal{Y}_N^A \cap \mathcal{Y}_N|}{|\mathcal{Y}_N|}$. As a consequence, an approximate nondominated frontier with larger cardinality is more desirable.

- **Coverage.** A simple coverage indicator can be defined as

$$f^a := \frac{\sum_{\boldsymbol{y} \in \mathcal{Y}_N \setminus \mathcal{Y}_N^A} d(k(\boldsymbol{y}), \boldsymbol{y})}{|\mathcal{Y}_N \setminus \mathcal{Y}_N^A|},$$

  i.e., the average distance to the closest point in the approximate frontier. Observe that f^a can be viewed as a measure of dispersion of the points in the nondominated frontier. Smaller values of f^a indicate that the nondominated points in the approximate frontier are in different parts of the criterion space. As a consequence, an approximate nondominated frontier with smaller f^a is more desirable.

- **Uniformity.** A uniformity indicator should capture how well the points in an approximate frontier are spread out. Points in a cluster do not increase the quality of an approximate frontier. We define the uniformity indicator to be

$$\mu := \frac{\sum_{\boldsymbol{y} \in \mathcal{Y}_N^A} n(\boldsymbol{y})}{|\mathcal{Y}_N^A|}.$$

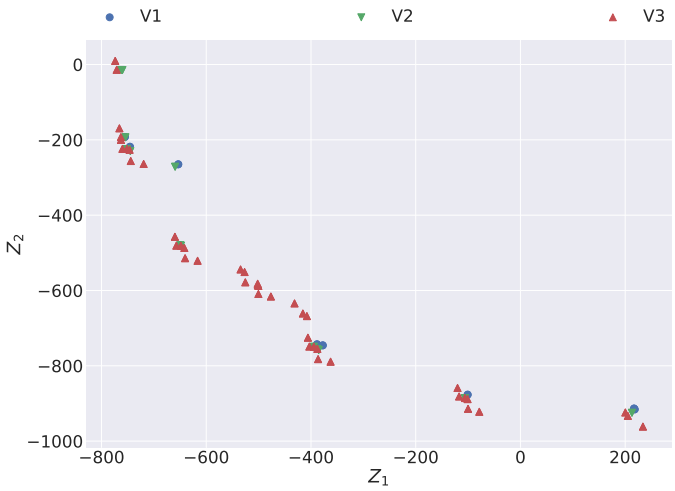
  As a consequence, an approximate nondominated frontier with smaller μ is more desirable.

7.1 MOMILPs with two objective functions

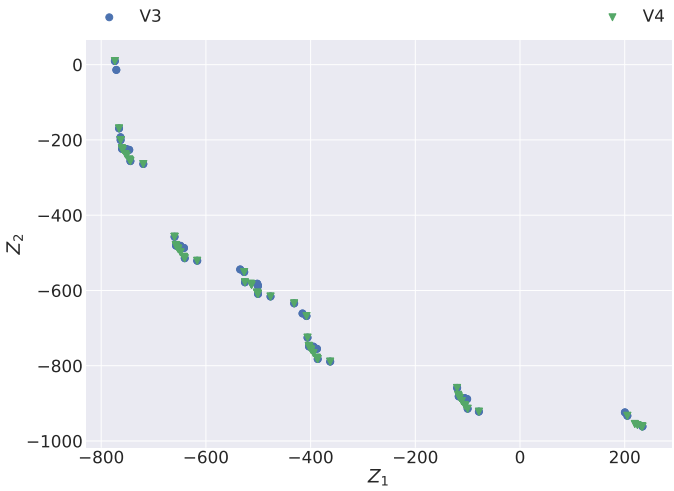
In this section, we compare the performance of V1 to V5T4 on 37 instances of MOMILPs with $p = 2$, i.e., instances of Mixed Binary and Facility Location problems listed in Table 1. A run time limit of 2 minutes is imposed for all experiments in this section. It is worth mentioning that large instances in this section may take over an hour to be solved by an exact method, the so-called the Triangle Splitting Method [5].

Figure 5 shows the approximate nondominated frontiers obtained by V1, V2, V3, V4, V5T1 and the true nondominated frontier for one of 37 instances in this section. Observe that the approximate nondominated frontier of V2 is better than V1, and V3 is better than V2, V4 is better than V3, and V5T1 is better than V4. Observe too that V5T1 has captured the form of the true nondominated frontier.

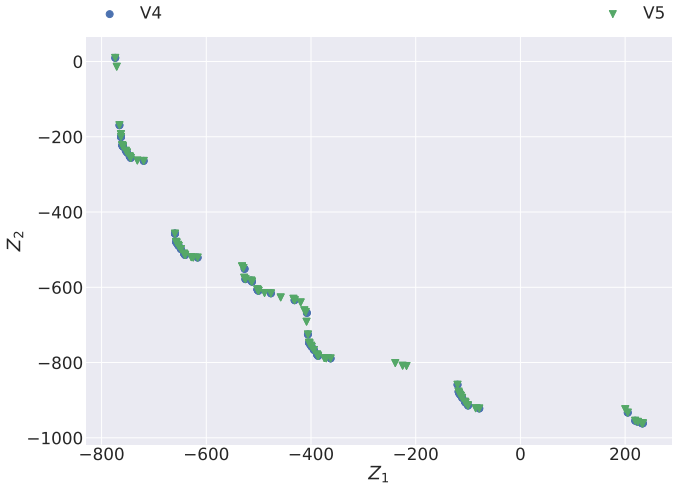
Figure 6 shows the performance of V1 to V5T4 for all 37 instances. In Figure 6a, we observe that, on average, V1, V2, and V3 are terminated in a fraction of a second. Not surprisingly, in Figure 6b, we observe that more points exist in the approximate nondominated frontier of the sophisticated versions in general. Overall, the existence of more points implies better approximations but sometimes this is not true and this is exactly what we observe by comparing V4 and V5T1 (in Figure 6b). In particular, we observe from Figure 6c-6e that the approximate nondominated frontier generated by V5T1 reaches a better value for the hypervolume gap, coverage, and uniformity indicators in comparison with V1 to V4. It is worth mentioning that we have not included the 'cardinality indicator' in this section since there exists infinite number of nondominated points in the true nondominated frontier of a MOMILP (in general). Finally, we note that the quality of approximate nondominated frontiers generated by FPBH becomes slightly better by activating parallelization option and increasing the number of threads. For example, we see from Figure 6c that the hypervolume gap is less than 0.5% on average for V5T4.



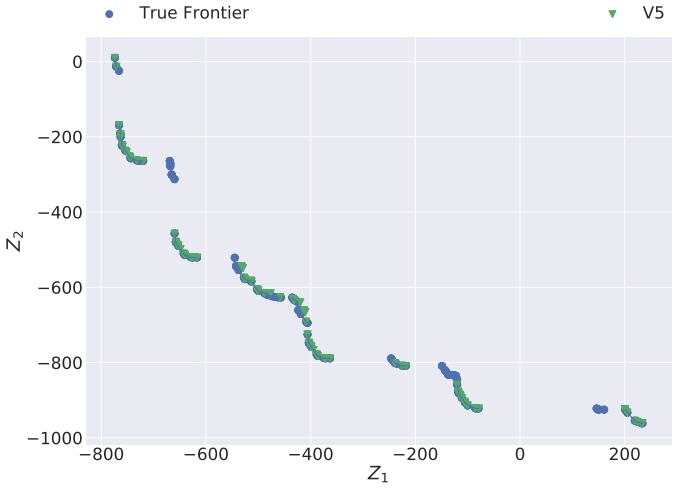
(a) V1 vs V2 vs V3



(b) V3 vs V4

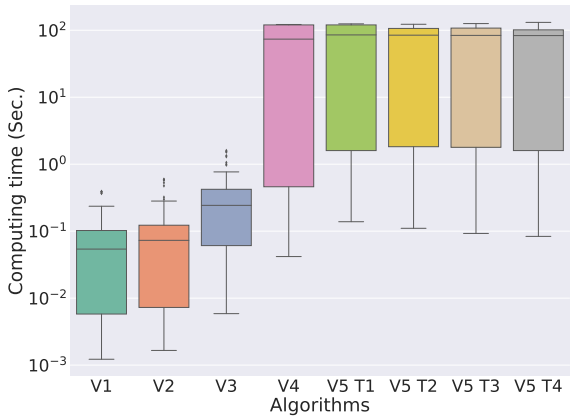


(c) V4 vs V5T1

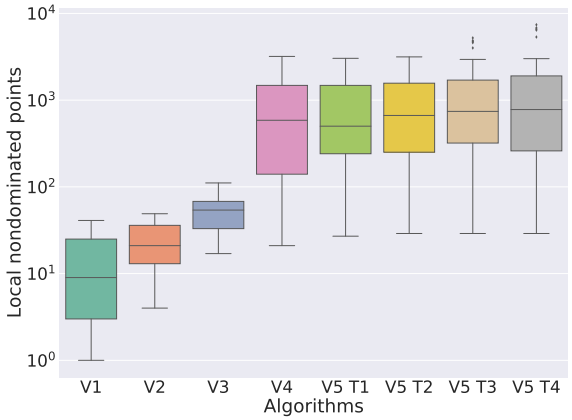


(d) V5T1 vs the true nondominated frontier

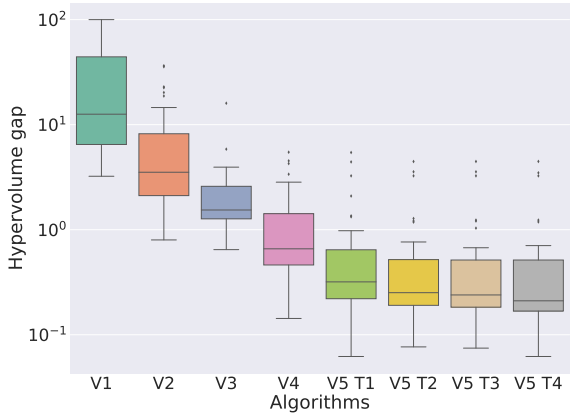
Figure 5: A small example (Instance #18 in [5])



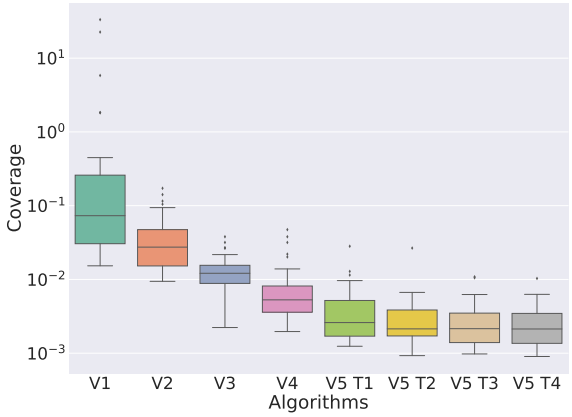
(a) Solution time



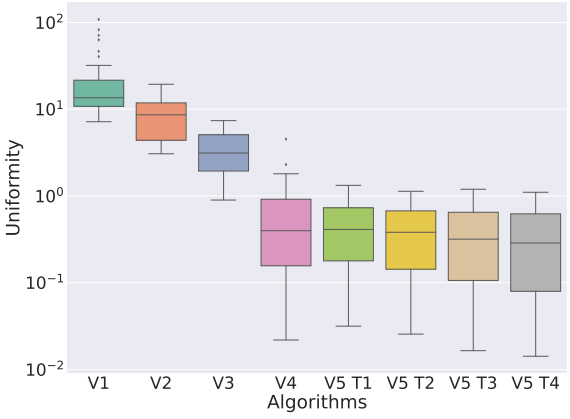
(b) The number of points



(c) Hypervolume gap

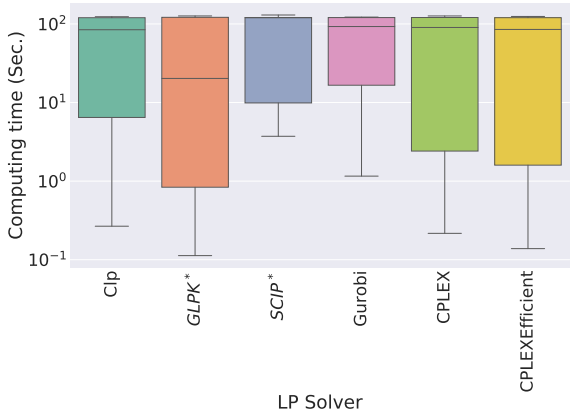


(d) Coverage

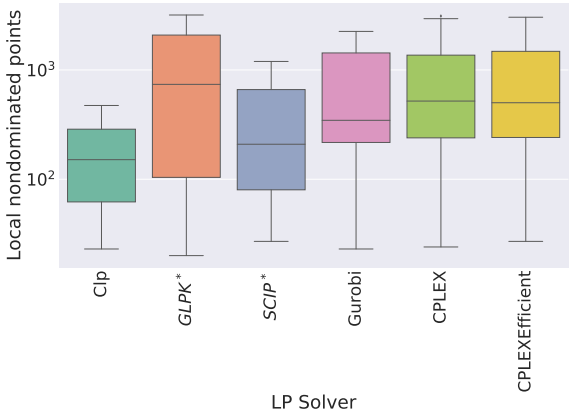


(e) Uniformity

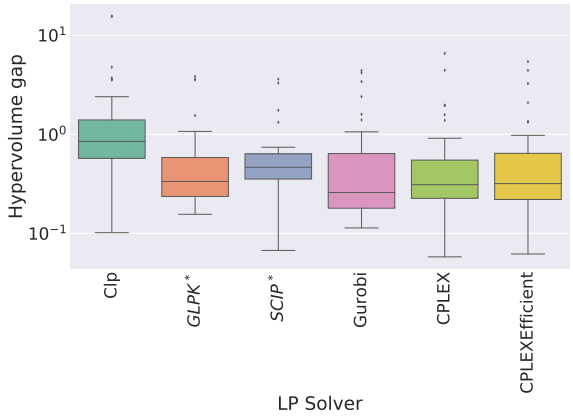
Figure 6: Performance of V1 to V5T4 on MOMILPs with two objectives



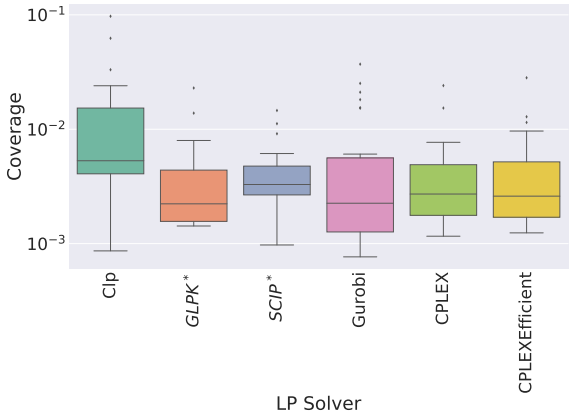
(a) Solution time



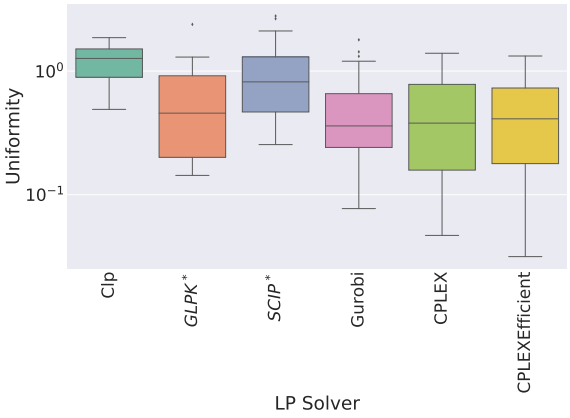
(b) The number of points



(c) Hypervolume gap



(d) Coverage



(e) Uniformity

Figure 7: Performance of V5T1 using different linear programming solvers on MOMILPs with two objectives

We now show that the effect of using different linear programming solvers for solving linear programs arising during the course of the FPBH. As mentioned earlier, we have two implementations of FPBH, one that supports any LP solver (including CPLEX) through MathProgBase.jl [27, 16] and the other implemented specifically for CPLEX using CPLEX.jl, denoted as CPLEXEfficient. Figure 7 shows the performance of V5T1 on all 37 instances for FPBH with different LP solvers (CPLEX 12.7, Gurobi 7.5, SCIP 4.0, Clp 1.15, and GLPK 4.61) and for the efficient implementation of FPBH with CPLEX.jl. Overall, from Figure 7, we observe that all solvers are competitive but the commercial solvers including CPLEX and Gurobi generate slightly better approximations with respect to all quality indicators. We also note that in Figure 7, there is a 'star symbol' in front of the terms GLPK and SCIP since these two solvers went out of memory when solving instances of the facility location problem. So, the boxplots corresponding to these solvers do not contain instances of the facility location problem.

7.2 MOMILPs with no continuous variables

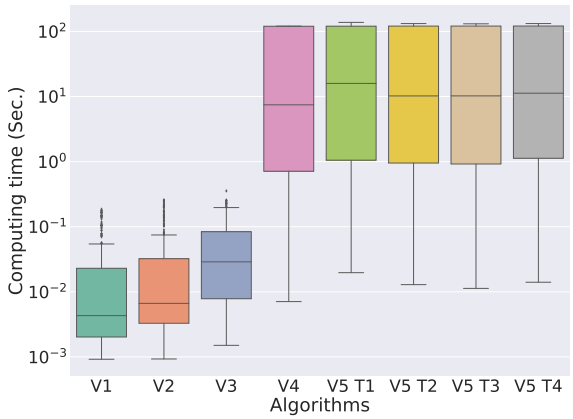
In this section, we compare the performance of V1 to V5T4 on 254 instances of MOMILPs with $p > 2$ and $n_2 = 0$, i.e., instances of knapsack and assignment problems listed in Table 1. A run time limit of 2 minutes is imposed for all experiments in this section. It is worth mentioning that large instances in this section may take several hours to be solved by an exact method [23]. We also do not consider any instance of MOMILPs with $p = 2$ and $n_2 = 0$ in this section since we have shown in our previous study that a feasibility pump based algorithm works well for such instances [31].

Figure 8 shows the performance of V1 to V5T4 for all 254 instances. The quality of approximate nondominated frontiers generated can be clearly observed from Figure 8 with respect to all quality indicators. The cardinality indicator shows that around 40% of the true nondominated points are computed using V5T1. Also, the quality of approximate nondominated frontiers generated by FPBH becomes slightly better by activating parallelization option and increasing the number of threads. For example, we see from Figure 8c that the hypervolume gap is less than 1% on average for V5T4.

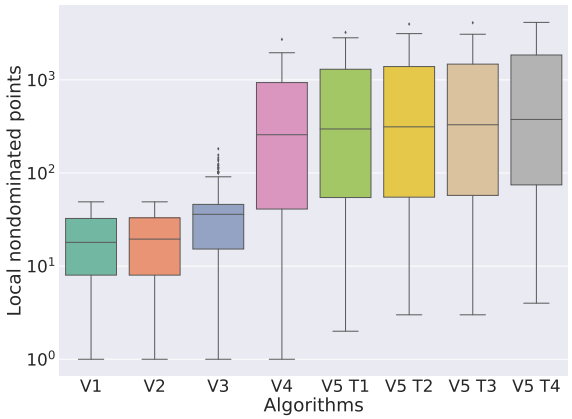
Figure 9 shows the performance of V5T1 on all 254 instances when the algorithm is built at the top of different linear programming solvers. Overall, we observe that the solvers are competitive but 'CPLEXEfficient' generates better approximations with respect to all quality indicators.

We now compare the performance of V1 to V5T1 with MDLS [39]. For this comparison, we have used the C++ implementation of MDLS which is publicly available at `http://prolog.univie.ac.at/research/MDLS`. It is worth mentioning that this implementation is problem-dependent in a sense that some of its source/header files should be specifically defined for any problem that we are trying to solve. Fortunately, we are able to employ MDLS for all 154 instances of knapsack problem . This is because in the available implementation, the corresponding header/source files for solving such instances exist.

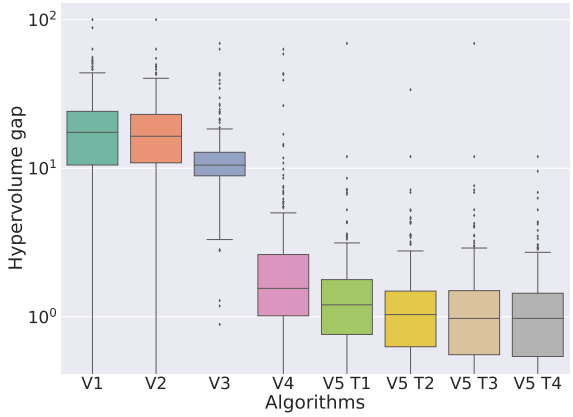
We use the default setting of MDLS in which it repeats for 10 runs for each instance, and reports the results of each run. For comparison purposes, we accumulate the results of all 10 runs for each instance. We again impose a time limit of 120 seconds for different version of our algorithm but do not impose any time limit for MDLS since it naturally terminates within 120 seconds. Figure 10 show the performance of V1 to V5T1 on the instances of the knapsack problem. Overall, we see that even V1 outperforms MDLS.



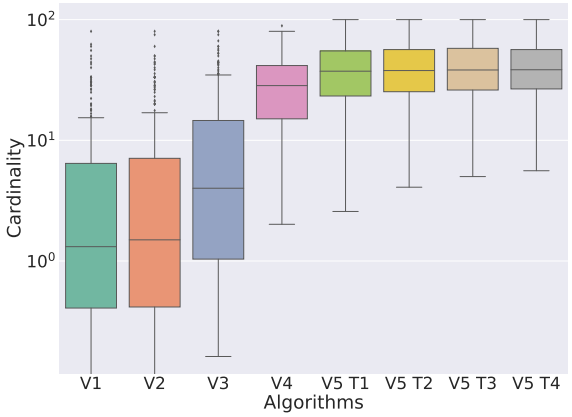
(a) Solution time



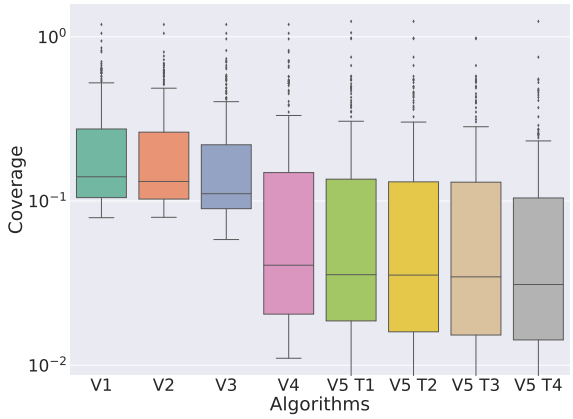
(b) The number of points



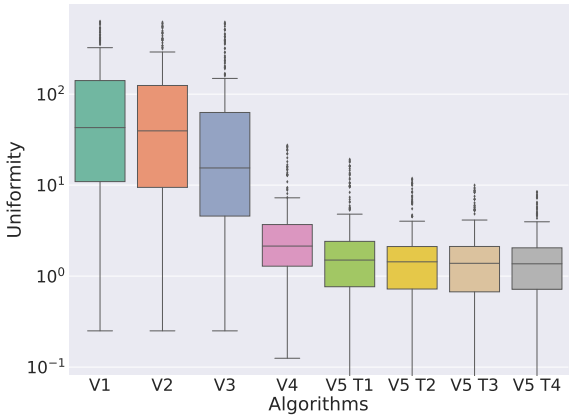
(c) Hypervolume gap



(d) Cardinality

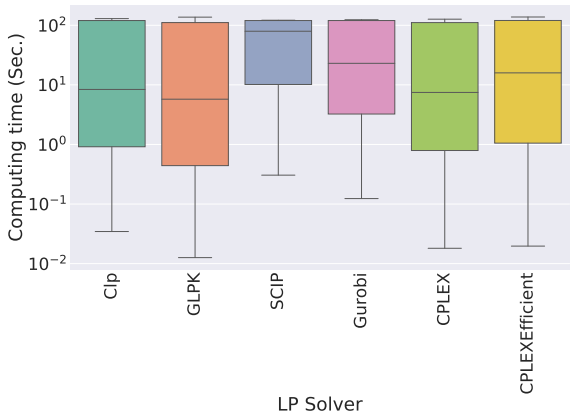


(e) Coverage

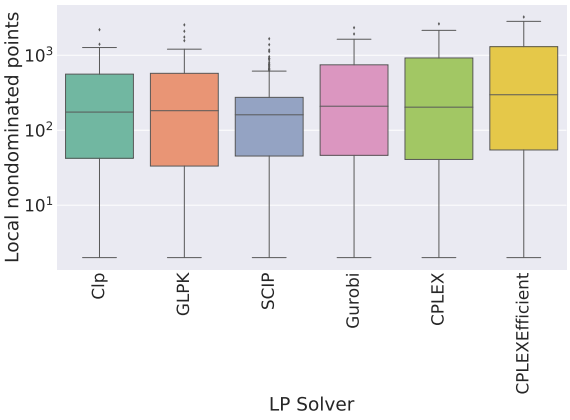


(f) Uniformity

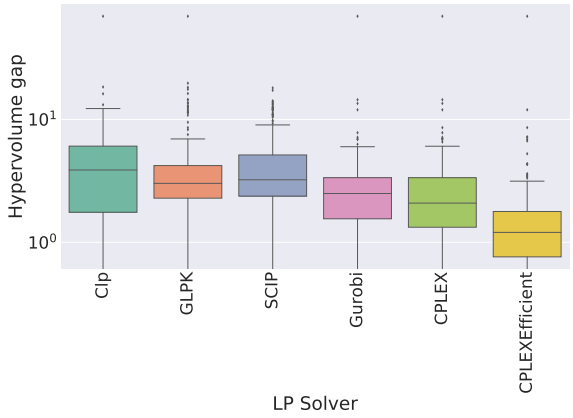
Figure 8: Performance of V1 to V5T4 on MOMILPs with no continuous variables



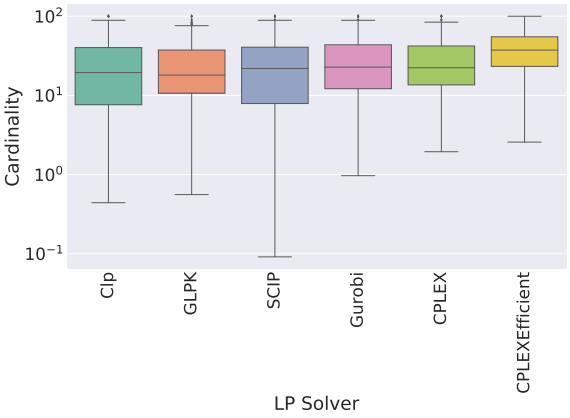
(a) Solution time



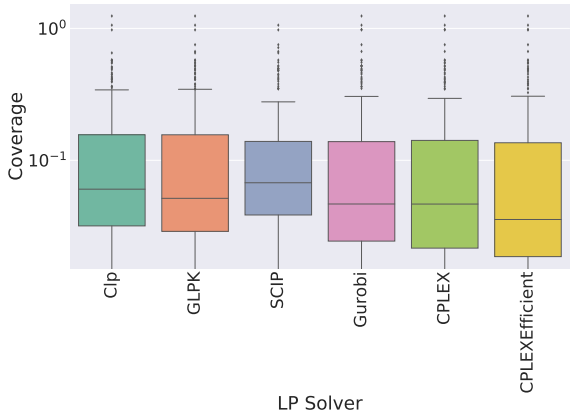
(b) The number of points



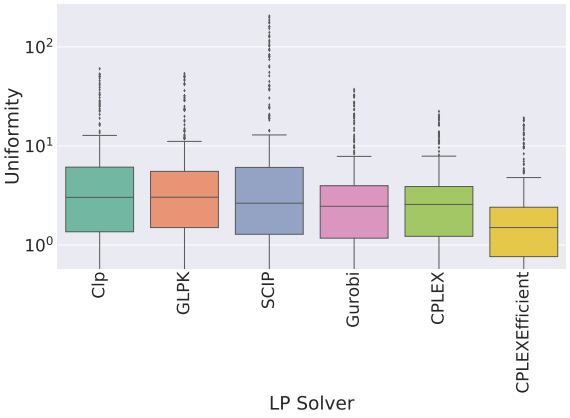
(c) Hypervolume gap



(d) Cardinality

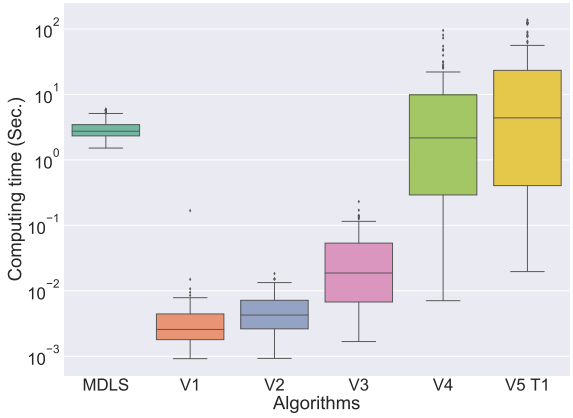


(e) Coverage

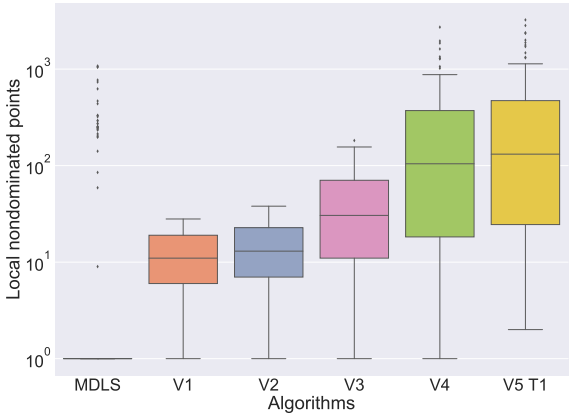


(f) Uniformity

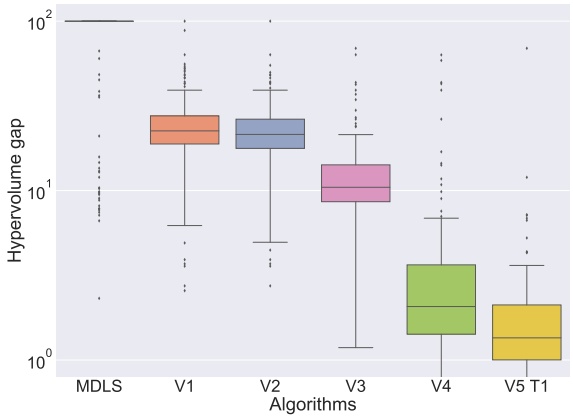
Figure 9: Performance of V5T1 using different linear programming solvers on MOMILPs with no continuous variable



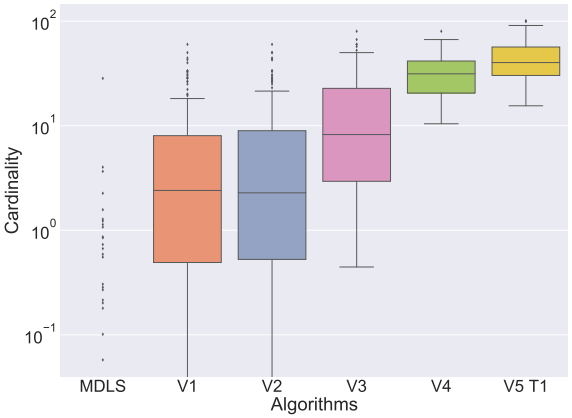
(a) Solution time



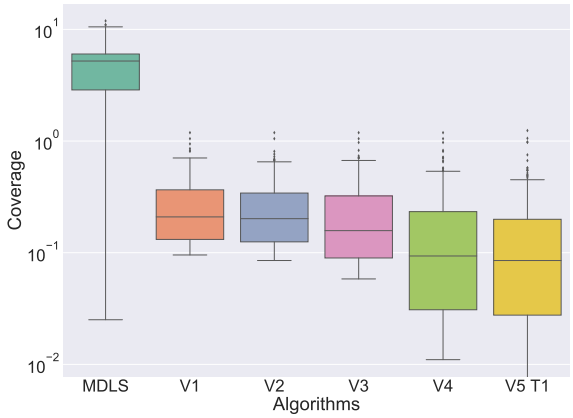
(b) The number of points



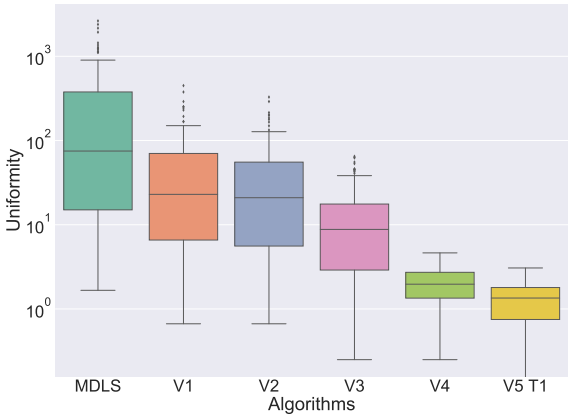
(c) Hypervolume gap



(d) Cardinality

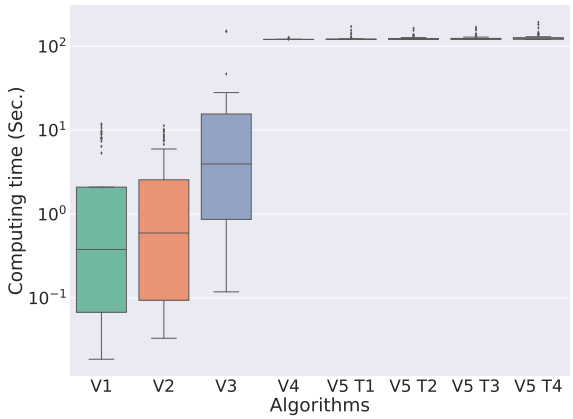


(e) Coverage

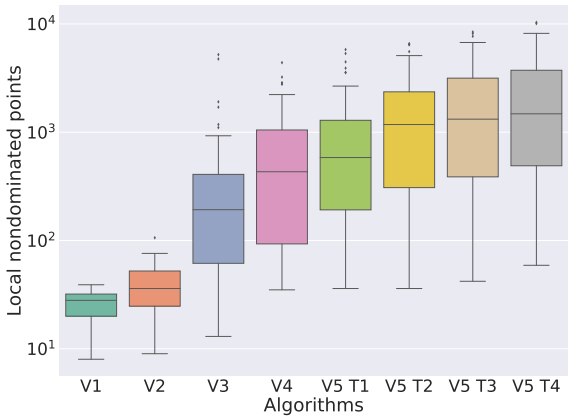


(f) Uniformity

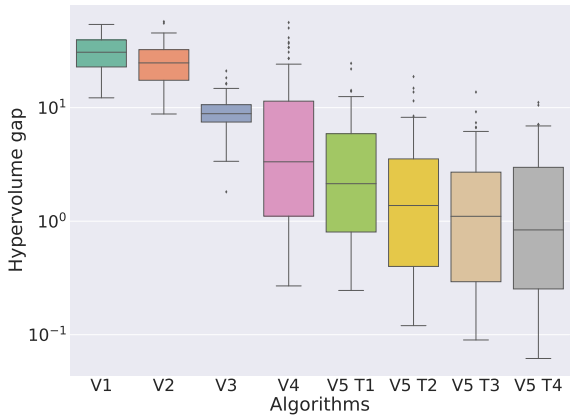
Figure 10: Performance comparison of V1 to V5T1 with MDLS on MOMILPs with no continuous variable



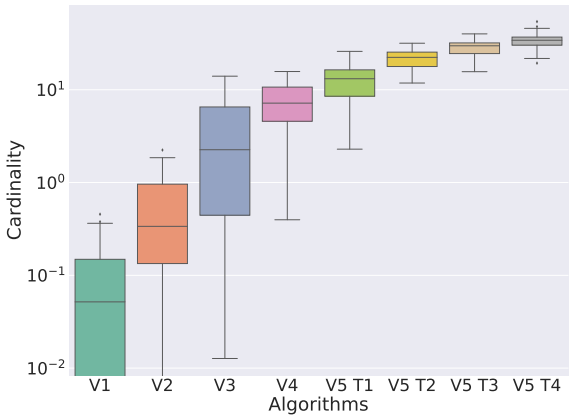
(a) Solution time



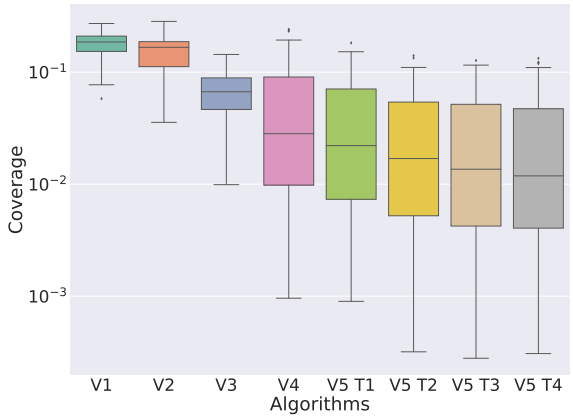
(b) The number of points



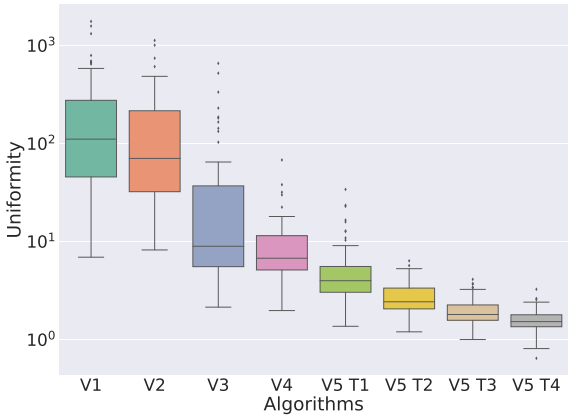
(c) Hypervolume gap



(d) Cardinality

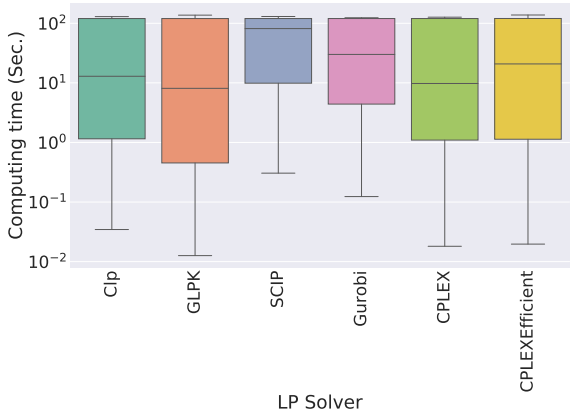


(e) Coverage

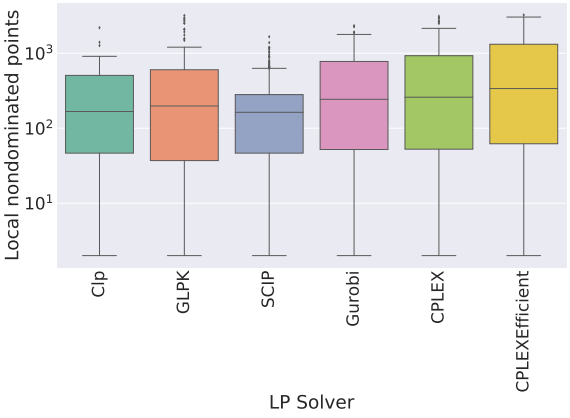


(f) Uniformity

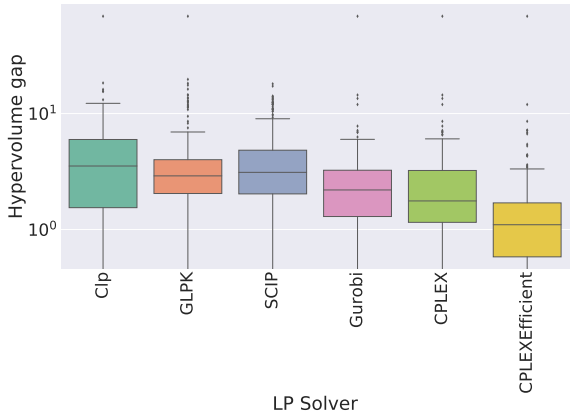
Figure 11: Performance V1 to V5T4 on MOMILPs with more than two objectives



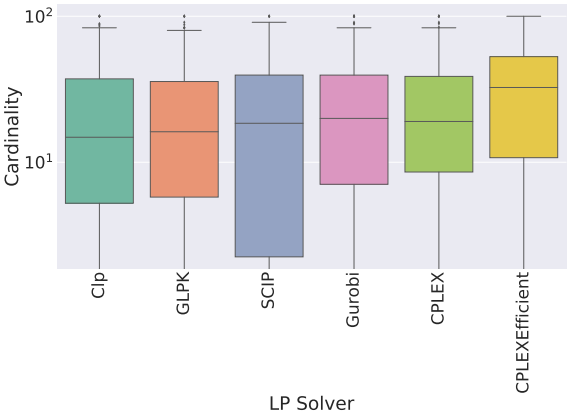
(a) Solution time



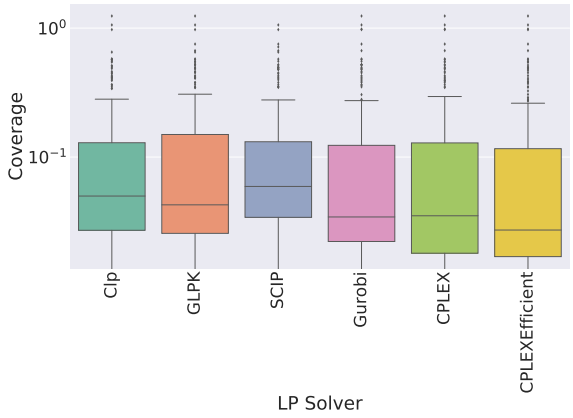
(b) The number of points



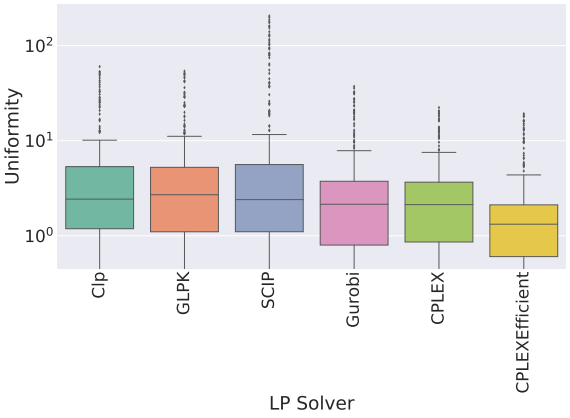
(c) Hypervolume gap



(d) Cardinality



(e) Coverage



(f) Uniformity

Figure 12: Performance of V5T1 using different linear programming solvers on MOMILPs with more than two objectives

7.3 MOMILPs with more than two objective functions

In this section, we compare the performance of V1 to V5T4 on 60 instances of MOMILPs with $p > 2$ and $n_1, n_2 > 0$, i.e., instances of mixed binary problems listed in Table 1. A run time limit of 2 minutes is imposed for all experiments in this section. We again note that (to the best of our knowledge) there exists no exact solution approach for solving instances in this section. So, for each instance, we run V5T1 for 15 minutes and treat its outputs as the true nondominated frontier.

Figure 11 shows the performance of V1 to V5T4 for all 60 instances. Again, we observe that using more sophisticated versions and more number of threads can result in generating better approximations. For example, we see from Figure 11c that the hypervolume gap is less than 1% on average for V5T4. Figure 12 shows the performance of V5T1 on all 5 instances when the algorithm is built at the top of different linear programming solvers. Again, we observe that the solvers are competitive but 'CPLEXEfficient' generates better approximations with respect to all quality indicators.

8 Final Remarks

We extended our recent algorithm for computing approximate nondominated frontiers of bi-objective pure integer linear programs to multi-objective mixed integer linear programs with an arbitrary number of objective functions. To the best of our knowledge, we are the first authors introducing a feasibility pump based heuristic for solving multi-objective mixed integer linear programs. In addition to the feasibility pump method, the proposed algorithm exploits the underlying ideas of several other algorithms in the literature of both single-objective and multi-objective optimization including a weighted sum method, a local search approach, and a local branching method. The algorithm also employs a decomposition technique for generating approximate nondominated points from different parts of the criterion space. We showed numerically that: (1) the performance of the algorithm is almost independent of the type of linear programming solver embedded in the algorithm; (2) The algorithm produces high-quality approximate nondominated frontiers in a short time; (3) It can naturally exploits parallelism and compute even better approximations if this option is activated; and (4) It outperforms MDLS on instances of multi-objective knapsack problem.

References

[1] Achterberg, T. and Berthold, T. (2007). Improving the feasibility pump. *Discrete Optimization*, 4(1):77 – 86. Mixed Integer ProgrammingIMA Special Workshop on Mixed-Integer Programming.

[2] Aneja, Y. P. and Nair, K. P. K. (1979). Bicriteria transportation problem. *Management Science*, 27:73–78.

[3] Berthold, T., Lodi, A., and Salvagnin, D. (2017). Ten years of feasibility pump and counting. *Preprint*. `http://cerc-datascience.polymtl.ca/wp-content/uploads/2017/08/Technical-Report_DS4DM-2017-009.pdf`.

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

[5] Boland, N., Charkhgard, H., and Savelsbergh, M. (2015b). A criterion space search algorithm for biobjective mixed integer programming: The triangle splitting method. *INFORMS Journal on Computing*, 27(4):597–618.

[6] Boland, N., Charkhgard, H., and Savelsbergh, M. (2016a). The L-shape search method for triobjective integer programming. *Mathematical Programming Computation*, 8(2):217–251.

[7] Boland, N., Charkhgard, H., and Savelsbergh, M. (2016b). The quadrant shrinking method: A simple and efficient algorithm for solving tri-objective integer programs. *European Journal of Operational Research*. available online.

[8] Boland, N. L., Eberhard, A. C., Engineer, F. G., Fischetti, M., Savelsbergh, M. W. P., and Tsoukalas, A. (2014). Boosting the feasibility pump. *Mathematical Programming Computation*, 6(3):255–279.

[9] Bringmann, K. and Friedrich, T. (2010). Approximating the volume of unions and intersections of high-dimensional geometric objects. *Computational Geometry*, 43(6):601 – 610.

[10] Castillo Tapia, M. and Coello Coello, C. A. (2007). Applications of multi-objective evolutionary algorithms in economics and finance: A survey. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 532–539.

[11] Chalmet, L. G., Lemonidis, L., and Elzinga, D. J. (1986). An algorithm for bi-criterion integer programming problem. *European Journal of Operational Research*, 25:292–300.

[12] Coello Coello, C. A., Lamont, G. B., and Van Veldhuizen, D. A. (2007). *Evolutionary Algorithms for Solving Multi-Objective Problems*. Springer, New York.

[13] Dächert, K., Gorski, J., and Klamroth, K. (2012). An augmented weighted Tchebycheff method with adaptively chosen parameters for discrete bicriteria optimization problems. *Computers & Operations Research*, 39:2929–2943.

[14] Dächert, K. and Klamroth, K. (2014). A linear bound on the number of scalarizations needed to solve discrete tricriteria optimization problems. *Journal of Global Optimization*, pages 1–34.

[15] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197.

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

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

[18] Ehrgott, M. (2005). *Multicriteria optimization*. Springer Berlin Heidelberg, second edition.

[19] Ehrgott, M. and Gandibleux, X. (2004). Approximative solution methods for multiobjective combinatorial optimization. *Top*, 12(1):1–63.

[20] Fischetti, M., Glover, F., and Lodi, A. (2005). The feasibility pump. *Mathematical Programming*, 104(1):91–104.

[21] Fischetti, M. and Lodi, A. (2003). Local branching. *Mathematical Programming*, 98(1):23–47.

[22] Fischetti, M. and Salvagnin, D. (2009). Feasibility pump 2.0. *Mathematical Programming Computation*, 1(2):201–222.

[23] Kirlik, G. and 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.

[24] Köksalan, M. and Lokman, B. (2014). Finding nadir points in multi-objective integer programs. *Journal of Global Optimization*, pages 1–23.

[25] Lei, D. (2009). Multi-objective production scheduling: a survey. *International Journal of Advanced Manufacturing Technology*, 43(9):926–938.

[26] Lian, K., Milburn, A. B., and Rardin, R. L. (2016). An improved multi-directional local search algorithm for the multi-objective consistent vehicle routing problem. *IIE Transactions*, 48(10):975–992.

[27] Lubin, M. and Dunning, I. (2015). Computing in operations research using Julia. *INFORMS Journal on Computing*, 27(2):238–248.

[28] Lust, T. and Teghem, J. (2010). The multiobjective traveling salesman problem: A survey and a new approach. In Coello Coello, C. A., Dhaenens, C., and Jourdan, L., editors, *Advances in Multi-Objective Nature Inspired Computing*, volume 272 of *Studies in Computational Intelligence*, pages 119–141. Springer Berlin Heidelberg.

[29] Özlen, M., Burton, B. A., and MacRae, C. A. G. (2013). Multi-objective integer programming: An improved recursive algorithm. *Journal of Optimization Theory and Applications*. 10.1007/s10957-013-0364-y.

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

[31] Pal, A. and Charkhgard, H. (2017). A feasibility pump and local search based heuristic for bi-objective pure integer linear programming. *Preprint*. `http://www.optimization-online.org/DB_FILE/2017/03/5902.pdf`.

[32] Paquete, L., Schiavinotto, T., and Stützle, T. (2007). On local optima in multiobjective combinatorial optimization problems. *Annals of Operations Research*, 156(1):83.

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

[34] Przybylski, A., Gandibleux, X., and Ehrgott, M. (2010a). A two phase method for multi-objective integer programming and its application to the assignment problem with three objectives. *Discrete Optimization*, 7(3):149 – 165.

[35] Przybylski, A., Gandibleux, X., and Ehrgott, M. (2010b). A two phase method for multi-objective integer programming and its application to the assignment problem with three objectives. *Discrete Optimization*, 7(3):149 – 165.

[36] Requejo, C. and Santos, E. (2017). *A Feasibility Pump and a Local Branching Heuristics for the Weight-Constrained Minimum Spanning Tree Problem*, pages 669–683. Springer International Publishing, Cham.

[37] Sayın, S. (2000). Measuring the quality of discrete representations of efficient sets in multiple objective mathematical programming. *Mathematical Programming*, 87(3):543–560.

[38] Soylu, B. and Yıldız, G. B. (2016). An exact algorithm for biobjective mixed integer linear programming problems. *Computers & Operations Research*, 72:204 – 213.

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

[40] Zhou, A., Qu, B.-Y., Li, H., Zhao, S.-Z., Suganthan, P. N., and Zhang, Q. (2011). Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm and Evolutionary Computation*, 1(1):32–49.

[41] Zitzler, E., Brockhoff, D., and Thiele, L. (2007). The hypervolume indicator revisited: On the design of Pareto-compliant indicators via weighted integration. In Obayashi, S., Deb, K., Poloni, C., Hiroyasu, T., and Murata, T., editors, *Evolutionary Multi-Criterion Optimization*, volume 4403 of *Lecture Notes in Computer Science*, pages 862–876.

[42] Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C., and Grunert da Fonseca, V. (2003). Performance assessment of multiobjective optimizers: an analysis and review. *Evolutionary Computation, IEEE Transactions on*, 7(2):117–132.