`OOES.jl`: A julia package for optimizing a linear function over the set of efficient solutions for bi-objective mixed integer linear programming

Alvaro Sierra-Altamiranda, Hadi Charkhgard

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

Abstract

We present `OOES.jl`, a package for optimizing a linear function over the efficient set of bi-objective mixed integer linear programs. The proposed package extends our recent study by adding two main features: (1) In addition to CPLEX, the package allows employing any single-objective solver supported by `MathProgBase.jl`, e.g., GLPK, CPLEX, and SCIP; (2) The package supports execution on multiple processors and is compatible with `JuMP` modeling language. An extensive computational study shows the efficacy of the package and its features.

Keywords: bi-objective mixed integer linear programming, optimization over efficient set, parallelization, commercial and non-commercial solvers

1. Introduction

Many real-world optimization problems can be formulated as mixed integer linear programs. These problems often involve multiple conflicting objectives, i.e., it is impossible to find a feasible solution that simultaneously optimizes all objectives. Consequently, the focus in multi-objective optimization has been primarily on developing algorithms for generating some (if not all) *efficient* solutions, i.e., solutions in which it is impossible to improve the value of one objective without a deterioration in the value of at least one other objective. Although understanding the trade-offs between objectives can help decision makers select their preferred solutions, some authors (see for instance [7]) argue that presenting too many efficient solutions may only confuse the decision makers(s). An approach that alleviates this issue is finding a preferred solution among the set of efficient solutions directly and is known as *optimizing over the efficient set*.

The problem of optimizing a linear function over the efficient set has a rich literature in multi-objective linear programming (see for instance [2, 11]). Also, in recent years, this problem has received attentions for multi-objective pure integer linear programs (see for instance [1, 3, 4, 5, 7]). However, no exact algorithm is known for optimizing a linear function over the set of efficient solutions of multi-objective mixed integer

linear programs involving more than two objectives. In fact, in our recent study [10], we proposed the first algorithm (to the best of our knowledge) for cases with two objectives, i.e., Bi-Objective Mixed Integer Linear Programs (BOMILPs). In light of this observation, in this study, we extend our previous work by creating a user-friendly open-source julia package which has the following additional desirable characteristics (compared to its original C++ implementation):

- The package benefits from the data structure of `FPBH.jl` [8], i.e., it is compatible with the popular `JuMP` modeling language [6] and supports input in LP and MPS file formats.
- The package supports execution on multiple processors and allows users to choose different parallelization techniques by just tunning a parameter. It is worth mentioning that several studies have been conducted about parallelization for evolutionary algorithms in multi-objective optimization (see for instance [8, 12]). However, unfortunately, this topic has been almost untouched in the literature of exact algorithms. The recent study conducted by Pettersson and Ozlen [9] is one of the few papers (if not the only one) in this scope.
- The package allows users to choose between different single-objective optimization solvers by just tunning a parameter. The default solvers include GLPK, CPLEX, Gurobi and SCIP, but it works for all other solvers supported by `MathProgBase.jl` as well.
- The package can be modified by users to return the entire nondominated frontier of a BOMILP.

In the remaining, our package is referred to as `OOES.jl` and the algorithm as `OOESAlg`. The structure of the paper is organized as follows: In Section 2, the main concepts and definitions are explained. In Section 3, we provide a brief explanation of `OOESAlg`. In Section 4, the main characteristics of the package are detailed. In Section 5, a comprehensive computational study is conducted. Finally, in Section 6, some concluding remarks are provided.

2. Preliminaries

A *Bi-Objective Mixed Integer Linear Program* (BOMILP) can be stated as follows:

$$\min_{(\boldsymbol{x}_I, \boldsymbol{x}_C) \in \mathcal{X}} \{z_1(\boldsymbol{x}_I, \boldsymbol{x}_C), z_2(\boldsymbol{x}_I, \boldsymbol{x}_C)\}, \tag{1}$$

where $\mathcal{X} := \left\{(\boldsymbol{x}_I, \boldsymbol{x}_C) \in \mathbb{Z}^{n_1}_{\geq} \times \mathbb{R}^{n_2}_{\geq} : A_1\boldsymbol{x}_I + A_2\boldsymbol{x}_C \leq \boldsymbol{b}\right\}$ 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 $\mathcal{X}$ is *bounded* and $z_i(\boldsymbol{x}_I, \boldsymbol{x}_C) = \boldsymbol{c}^\intercal_{i,I}\boldsymbol{x}_I + \boldsymbol{c}^\intercal_{i,C}\boldsymbol{x}_C$ where $\boldsymbol{c}_{i,I} \in \mathbb{R}^{n_1}$ and $\boldsymbol{c}_{i,C} \in \mathbb{R}^{n_2}$ for $i = 1, 2$ represents a linear objective function. The image $\mathcal{Y}$ of $\mathcal{X}$ under vector-valued function $\boldsymbol{z} := (z_1, z_2)^\intercal$ represents the *feasible set in the objective/criterion space*, that is $\mathcal{Y} := \{\boldsymbol{o} \in \mathbb{R}^2 : \boldsymbol{o} = \boldsymbol{z}(\boldsymbol{x}_I, \boldsymbol{x}_C) \text{ for all } (\boldsymbol{x}_I, \boldsymbol{x}_C) \in \mathcal{X}\}$.

A feasible solution $(\boldsymbol{x}_I, \boldsymbol{x}_C) \in \mathcal{X}$ is called *efficient* if there is no other $(\boldsymbol{x}'_I, \boldsymbol{x}'_C) \in \mathcal{X}$ such that $z_1(\boldsymbol{x}'_I, \boldsymbol{x}'_C) \leq z_1(\boldsymbol{x}_I, \boldsymbol{x}_C)$ and $z_2(\boldsymbol{x}'_I, \boldsymbol{x}'_C) < z_2(\boldsymbol{x}_I, \boldsymbol{x}_C)$ or $z_1(\boldsymbol{x}'_I, \boldsymbol{x}'_C) < z_1(\boldsymbol{x}_I, \boldsymbol{x}_C)$ and $z_2(\boldsymbol{x}'_I, \boldsymbol{x}'_C) \leq z_2(\boldsymbol{x}_I, \boldsymbol{x}_C)$. If $(\boldsymbol{x}_I, \boldsymbol{x}_C)$ is efficient, then $\boldsymbol{z}(\boldsymbol{x}_I, \boldsymbol{x}_C)$ is called a *nondominated point*. The set of all efficient solutions is denoted by $\mathcal{X}_E$. The set of all nondominated points $\boldsymbol{z}(\boldsymbol{x}_I, \boldsymbol{x}_C)$ for $(\boldsymbol{x}_I, \boldsymbol{x}_C) \in \mathcal{X}_E$ is denoted by $\mathcal{Y}_N$ and referred to as the *nondominated frontier*.

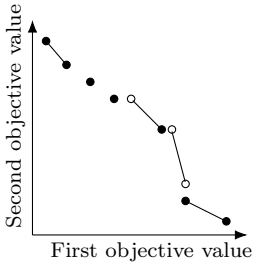


Figure 1: An example of the nondominated frontier of a BOMILP

Note that if $n_1 > 0$ and $n_2 > 0$, the nondominated frontier of a BOMILP can be complicated since it may contain isolated points as well as closed, half-open, and open segments. An example of a nondominated frontier of a BOMILP is shown in Figure 1. Although we assumed that $\mathcal{X}$ is bounded, we observe that the nondominated frontier of a BOMILP may still contain an infinite number of nondominated points (because of the existence of continuous segments) and so computing a (finite) exact representation of the nondominated frontier is quite challenging.

In light of the above, the problem of optimizing a linear function over the set of efficient solutions of a BOMILP can be stated as follows:

$$\min_{(\boldsymbol{x}_I, \boldsymbol{x}_C) \in \mathcal{X}_E} \quad f(\boldsymbol{x}_I, \boldsymbol{x}_C) \tag{2}$$

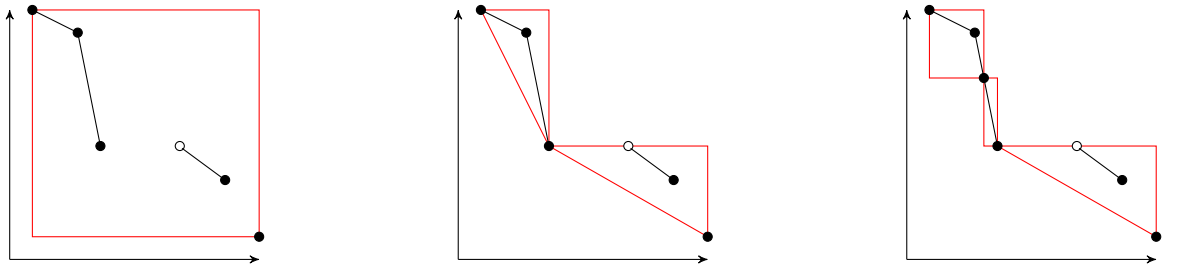
where $f(\boldsymbol{x}_I, \boldsymbol{x}_C) = \boldsymbol{c}^\intercal_{f,I}\boldsymbol{x}_I + \boldsymbol{c}^\intercal_{f,C}\boldsymbol{x}_C$ with $\boldsymbol{c}_{f,I} \in \mathbb{R}^{n_1}$ and $\boldsymbol{c}_{f,C} \in \mathbb{R}^{n_2}$ represents a linear function. To ensure that the problem cannot be solved straightforwardly, we assume that $\mathcal{X} \neq \mathcal{X}_E$ and $f(\boldsymbol{x})$ is not a strictly positive linear combination of $z_1(\boldsymbol{x})$ and $z_2(\boldsymbol{x})$. The following observation is helpful.

Observation 1. *Let $\mathcal{S} \subseteq \mathcal{X}$ be any arbitrary subset of the feasible solutions in the decision space such*

that $\mathcal{S}_E \subseteq \mathcal{X}_E$ where $\mathcal{S}_E$ is the set of efficient solutions of $\mathcal{S}$. Now let $f^l := \min_{(\boldsymbol{x}_I, \boldsymbol{x}_C) \in \mathcal{S}} f(\boldsymbol{x}_I, \boldsymbol{x}_C)$ and $f^u := \min_{(\boldsymbol{x}_I, \boldsymbol{x}_C) \in \mathcal{S}_E} f(\boldsymbol{x}_I, \boldsymbol{x}_C)$. If $\mathcal{S} = \mathcal{X}$, then f^l is a global lower (or dual) bound and f^u is a global upper (or primal) bound for the optimal value of Problem (2)*. Otherwise, i.e., $\mathcal{S} \neq \mathcal{X}$, f^l and f^u are local lower bound and local upper bound for the optimal value of Problem* (2)*, respectively.*

3. The algorithm

In our previous study [10], we presented the first criterion space search algorithm, referred to as `OOESAlg`, for optimizing a linear function over the set of efficient solutions of BOMILPs.



(a) Finding endpoints (b) Generated triangles (c) Generated rectangles

Figure 2: Progression of `OOESAlg` in terms of the discovery of nondominated points

`OOESAlg` maintains a priority queue of rectangles and right-angled triangles (defined by two nondominated points in the criterion space) that need to be explored. At the beginning, this priority queue is empty, so `OOESAlg` first computes the endpoints of the nondominated frontier. These two points are then used to define the first rectangle containing all the "not-yet-found" nondominated points, as shown in Figure 2a. Whenever the algorithm defines a rectangle, local lower and upper bounds are computed for the rectangle based on Observation 1. Note that computing a local upper bound is done based on the discovered nondominated points in a rectangle, e.g., for the first rectangle, the only discovered nondominated points are the endpoints.

`OOESAlg` explores a rectangle by finding the (locally) extreme nondominated points within the rectangle. It can be shown that by finding these points, the rectangle can be split into a set of right-angled triangles containing all "not-yet-found" nondominated points (see Figure 2b). Consequently, any triangle can be considered as a child node of a rectangle, and so the local lower bound of a triangle will be set equal to the local lower bound of its parent rectangle. In other words, the algorithm does not use its valuable computational time to compute a better local lower bound for a triangle (based on Observation 1). Overall, `OOESAlg` explores a triangle by first checking whether its hypotenuse is part of the nondominated frontier. If that is the case, then a local upper bound is computed over the hypotenuse of the triangle based on

Observation 1. Otherwise the triangle is split into at most two other rectangles, as shown in Figure 2c. The algorithm then adds those rectangles to the priority queue.

The priority queue maintains its elements in non-increasing order of their local lower bounds. Consequently, the local lower bound corresponding to the first element of the priority queue is always the global lower bound. Also, the algorithm keeps track of the global upper bound at any time by recording the best local upper bound found. `OOESAlg` reports an optimal solution if the absolute or relative gap between the global lower bound and the global upper bound is small enough at the beginning of an iteration or if the priority queue becomes empty. Note that if the priority queue becomes empty then the optimality gap is naturally zero. For a more detailed explanation of the algorithm, we refer the interested readers to [10].

4. Main characteristics of the package

In this section, we detail the main additional characteristics of `OOES.jl` compared to our previous C++ implementation of `OOESAlg`.

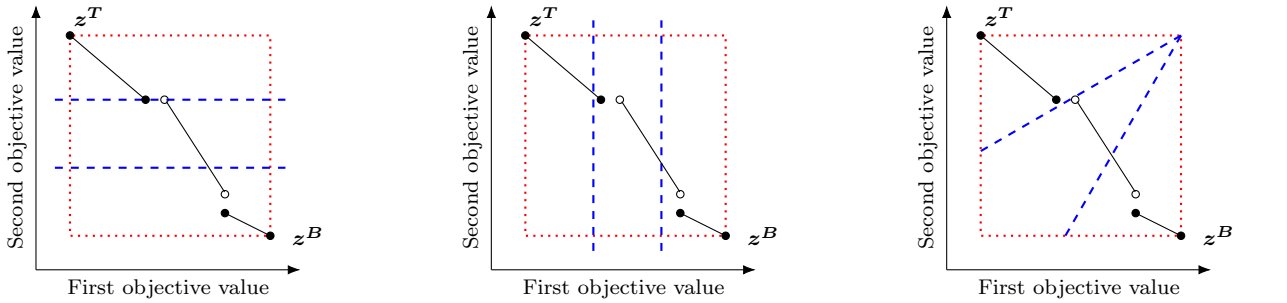
4.1. Parallelization techniques

`OOES.jl` benefits from the recent advances in modern computers, in terms of the number of processors, by exploiting parallelization. The package explores four different parallelization techniques, one of them is based on the priority queue, and the other three are based on decomposing the criterion space.

The simplest and most natural parallelization technique explores elements of the priority queue in parallel using different threads. Suppose that t is the number of available threads for parallelization and q is the number of elements in the priority queue. The first element of the priority queue is assigned to the first available thread to be explored, the second element to the second thread, and this procedure continues until t elements are assigned. We will later show in our numerical experiments that this technique maximizes the utilization of the available threads, i.e., improves the performance of the algorithm significantly.

The criterion space parallelization techniques are based on splitting the unexplored nondominated frontier between the endpoints by adding cuts. We consider three types of cuts to split the criterion space based on their directions including horizontal, vertical, and diagonal. An illustration of these cuts can be found in Figure 3 when $t = 3$ (note that the number of cuts is $t - 1$).

For horizontal splitting, the height of the nondominated frontier is divided by the number of threads that are available for parallelization, i.e., the distance between consecutive cuts is $\frac{z_2^T(\boldsymbol{x}) - z_2^B(\boldsymbol{x})}{t}$. The following



(a) Horizontal split (b) Vertical split (c) Diagonal split (origin: (z_1^B, z_2^T))

Figure 3: Splitting directions on the criterion space using three threads

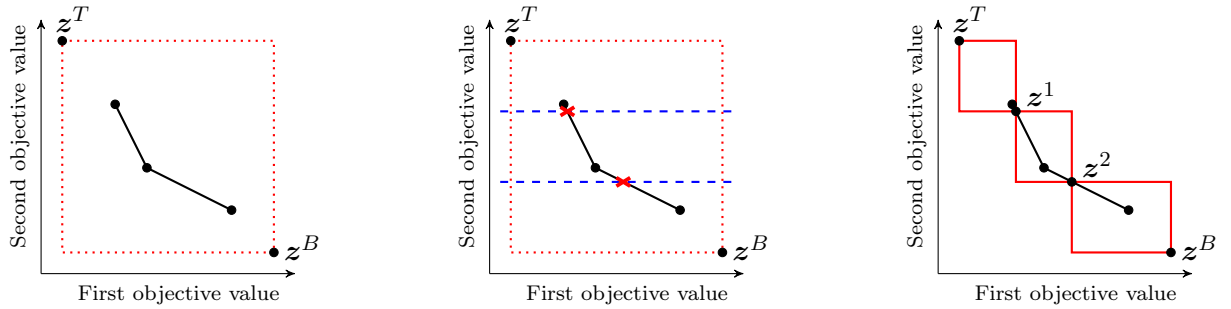
optimization problems are performed to find a nondominated point for each cut $v \in \{1, 2, ..., t-1\}$:

$$\tilde{\boldsymbol{z}}^v = \min_{(\boldsymbol{x}_I, \boldsymbol{x}_C) \in \mathcal{X}} \{z_1(\boldsymbol{x}_I, \boldsymbol{x}_C) : z_2(\boldsymbol{x}_I, \boldsymbol{x}_C) \leq z_2^B(\boldsymbol{x}) + \frac{v(z_2^T(\boldsymbol{x}) - z_2^B(\boldsymbol{x}))}{t}\},$$

followed by,

$$\boldsymbol{z}^v = \min_{(\boldsymbol{x}_I, \boldsymbol{x}_C) \in \mathcal{X}} \{z_1(\boldsymbol{x}_I, \boldsymbol{x}_C) + z_2(\boldsymbol{x}_I, \boldsymbol{x}_C) : \boldsymbol{z}(\boldsymbol{x}_I, \boldsymbol{x}_C) \leq \tilde{\boldsymbol{z}}^v\}.$$

Observe that, $\tilde{\boldsymbol{z}}^v$ may not be a nondominated point, therefore, the second operation is performed to find a nondominated point $\boldsymbol{z}^v$. Finally, the unexplored nondominated frontier is split into rectangles defined by every pair of consecutive nondominated points. An illustration of this technique using three threads can be found in Figure 4.



(a) The endpoints (b) Horizontal cuts (c) The generated rectangles

Figure 4: Horizontal splitting of the criterion space using three threads

The vertical splitting technique divides the width of the nondominated frontier by the number of available threads, i.e., the distance between consecutive cuts is $\frac{z_1^B(\boldsymbol{x}) - z_1^T(\boldsymbol{x})}{t}$. The following optimization problems are performed to find a nondominated point for each cut $v \in \{1, 2, ..., t-1\}$:

$$\tilde{\boldsymbol{z}}^v = \min_{(\boldsymbol{x}_I, \boldsymbol{x}_C) \in \mathcal{X}} \{z_2(\boldsymbol{x}_I, \boldsymbol{x}_C) : z_1(\boldsymbol{x}_I, \boldsymbol{x}_C) \leq z_1^T(\boldsymbol{x}) + \frac{v(z_1^B(\boldsymbol{x}) - z_1^T(\boldsymbol{x}))}{t}\},$$

followed by,

$$\boldsymbol{z}^v = \min_{(\boldsymbol{x}_I, \boldsymbol{x}_C) \in \mathcal{X}} \{z_1(\boldsymbol{x}_I, \boldsymbol{x}_C) + z_2(\boldsymbol{x}_I, \boldsymbol{x}_C) : \boldsymbol{z}(\boldsymbol{x}_I, \boldsymbol{x}_C) \leq \tilde{\boldsymbol{z}}^v\}.$$

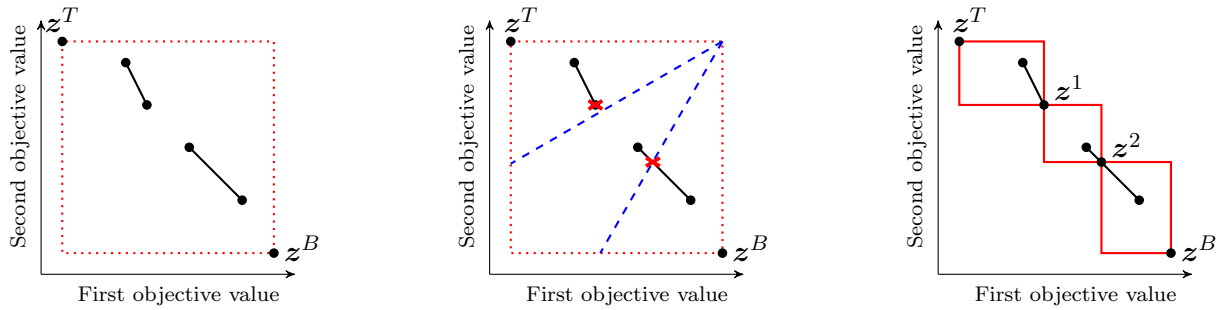
Finally, in the diagonal splitting, the algorithm attempts to divide the criterion space by adding cuts in which their slopes are from the set $\alpha \in \{\frac{\pi}{2t}, ..., \frac{(t-1)\pi}{2t}\}$ and originated from the point (z_1^B, z_2^T). The following optimization problems are performed to find a nondominated point for each cut $v \in \{1, 2, ..., t-1\}$:

$$\tilde{\boldsymbol{z}}^v = \min_{(\boldsymbol{x}_I, \boldsymbol{x}_C) \in \mathcal{X}} \{z_2(\boldsymbol{x}_I, \boldsymbol{x}_C) : z_2(\boldsymbol{x}_I, \boldsymbol{x}_C) - z_2^T \geq \tan(\frac{v\pi}{2t})(z_1(\boldsymbol{x}) - z_1^B)\},$$

followed by,

$$\boldsymbol{z}^v = \min_{(\boldsymbol{x}_I, \boldsymbol{x}_C) \in \mathcal{X}} \{z_1(\boldsymbol{x}_I, \boldsymbol{x}_C) + z_2(\boldsymbol{x}_I, \boldsymbol{x}_C) : \boldsymbol{z}(\boldsymbol{x}_I, \boldsymbol{x}_C) \leq \tilde{\boldsymbol{z}}^v\}.$$

The first optimization problem finds a feasible solution that minimize the second objective function above the line $z_2(\boldsymbol{x}_I, \boldsymbol{x}_C) - z_2^T = \tan(\frac{v\pi}{2t})(z_1(\boldsymbol{x}) - z_1^B)$. An example of this technique using three threads can be observed in Figure 5.



(a) The endpoints (b) Diagonal cuts (c) The generated rectangles

Figure 5: Diagonal splitting of the criterion space using three threads

It is worth mentioning that splitting the criterion space results in at most t independent unexplored rectangles. However, it is possible that different cuts return the same nondominated point, i.e., there may exist $v, w \in \{1, 2, ..., t-1\}$ with $v \neq w$ such that $\boldsymbol{z}^v = \boldsymbol{z}^w$. If that is the case then `OOES.jl` is only able to employ less than t processors. Overall, employing any of the criterion space parallelization techniques will lead to the exploration of at most t independent rectangles. Thus, `OOES.jl` will return at most t independent solutions for Problem (2). So, at the time of termination, `OOES.jl` chooses the one that has minimum value for $f(\boldsymbol{x})$.

4.2. Single-objective solvers

Our package benefits from the flexibility of working with julia and the well-known optimization library `MathProgBase.jl` that allows users to choose between different single-objective optimization solvers. The default solvers are GLPK, CPLEX, Gurobi and SCIP, but any solver supported by `MathProgBase.jl` can be employed. Note that, GLPK and SCIP are non-commercial solvers, which means that our package is not limited by license availability.

We also developed a second implementation of `OOES.jl` just for CPLEX solver using the package `CPLEX.jl`, which is an unofficial interface for using the solver from the julia language. Overall, `CPLEX.jl` is more efficient than `MathProgBase.jl` in terms of memory usage because during the course of the algorithm, it keeps the optimization model in the memory and allows to add and remove constraints. Thus, it does not have to rebuild the optimization models repetitively. The effect of using different single-objective optimization solvers is widely analyzed in our comprehensive computational study.

5. A computational study

We conduct a comprehensive computational study to evaluate the performance of `OOES.jl`. As mentioned in the previous section, we developed two implementations of `OOES.jl`, one using the package `MathProgBase.jl` and the other using `CPLEX.jl`. In the remaining, the first implementation is referred to by the name of the single-objective solver (GLPK 4.61, CPLEX 12.7, Gurobi 7.5, and SCIP 5.0.1) employed in an experiment (by the package). So, for example, `SCIP` refers to the first implementation when SCIP 5.0.1 is employed as a single-objective solver. The second implementation is referred to as `OOESCplex`. 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, and the RedHat Enterprise Linux 6.8 operating system. Our user-friendly open-source julia package can be found at `https://goo.gl/JfErXn`.

To test the performance of the package, we use the instances employed in our previous work [10]. Instances are divided into 5 classes based on their size. Each class is denoted by Cm where $m \in \{20, 40, 80, 160, 320\}$ is the number of constraints for each instance in that class. Each class contains 50 different instances and so, in total, 250 instances are tested during this computational study.

5.1. `OOES.jl` vs. C++ implementation

In this section, we show the overall performance of `OOES.jl` compared to our previous C++ implementation of `OOESAlg` [10]. Since the C++ implementation works with CPLEX, we use `OOESCplex` for comparison

in this section. In Table 1, we present a comparison between `OOESCplex` and the C++ implementation where 'Time (sec.)' is the solution time in seconds and '#MILP' is the number of (single-objective) mixed integer linear programs solved. The last two columns of the table show the percentage of decrease in the solution time and the number of (single-objective) MILPs solved by `OOESCplex` compared to the C++ implementation. Numbers are averages over 10 instances and bold numbers show the instances that `OOESCplex` has a better performance on them.

Table 1: Performance of the new algorithm in comparison to C++ implementation

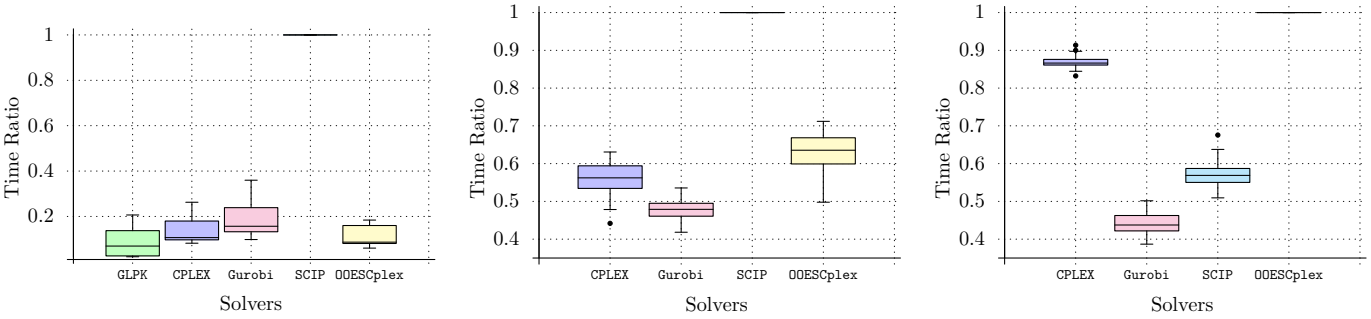
Class	OOESCplex.jl		C++		% Decrease	
	Time (sec.)	#MILP	Time (sec.)	#MILP	Time	#MILP
C20	0.14	85.40	0.23	85.50	**36.60%**	**0.12%**
	0.49	187.60	0.53	188.30	**7.10%**	**0.37%**
	0.52	166.90	0.62	171.60	**15.23%**	**2.74%**
	0.67	222.90	0.77	224.10	**12.75%**	**0.54%**
	0.16	87.80	0.19	87.80	**18.10%**	**0.00%**
Avg. C20	0.40	150.12	0.47	151.46	**14.90%**	**0.88%**
C40	5.41	723.80	6.15	726.00	**12.08%**	**0.30%**
	1.29	279.90	1.52	280.00	**15.21%**	**0.04%**
	1.93	311.60	2.13	312.60	**9.21%**	**0.32%**
	2.17	394.90	2.48	395.20	**12.63%**	**0.08%**
	2.37	382.70	2.71	383.50	**12.58%**	**0.21%**
Avg. C40	2.63	418.58	3.00	419.46	**12.17%**	**0.21%**
C80	42.63	1,757.30	41.00	1,757.30	-3.97%	**0.00%**
	20.83	1,017.50	20.57	1,017.40	-1.26%	-0.01%
	35.06	1,537.10	33.50	1,537.40	-4.63%	**0.02%**
	44.87	1,862.00	43.99	1,863.80	-2.01%	**0.10%**
	29.09	1,471.90	29.46	1,472.00	**1.26%**	**0.01%**
Avg. C80	34.49	1,529.16	33.70	1,529.58	-2.35%	**0.03%**
C160	269.57	2,457.40	243.29	2,459.00	-10.80%	**0.07%**
	282.94	2,133.60	249.77	2,145.50	-13.28%	**0.55%**
	255.31	2,216.40	229.59	2,213.00	-11.21%	-0.15%
	616.61	4,290.80	525.46	4,291.20	-17.35%	**0.01%**
	294.06	2,497.40	258.64	2,498.10	-13.70%	**0.03%**
Avg. C160	343.70	2,719.12	301.35	2,721.36	-14.05%	**0.08%**
C320	4,105.08	4,022.40	3,470.31	4,016.70	-18.29%	-0.14%
	6,544.82	6,363.70	5,556.96	6,363.90	-17.78%	**0.00%**
	4,417.13	4,329.10	3,732.42	4,332.00	-18.34%	**0.07%**
	5,446.13	4,955.90	4,542.32	4,956.90	-19.90%	**0.02%**
	3,873.17	4,108.00	3,294.50	4,110.50	-17.56%	**0.06%**
Avg. C320	4,877.27	4,755.82	4,119.30	4,756.00	-18.40%	**0.00%**

Note that `OOESCplex` solves slightly less single-objective mixed integer linear programs. This is due to some minor enhancements in the implementation (by removing redundant calculations). Moreover, observe that `OOESCplex` outperforms C++ implementation in small instances and remains competitive in the class $m = 80$. For larger instances, C++ implementation performs better (no more than 20%) because it uses the ILOG Concert Technology which is the official interface of CPLEX for C++ (but `CPLEX.jl` is not an official interface for julia). This interface is more efficient in terms of memory usage than `CPLEX.jl`.

5.2. Comparison between different solvers

In this section, we compare the performance of different solvers for `OOES.jl`. Figure 6 shows *the solution time ratios* of different solvers for different classes of instances. The solution time ratio is the ratio of the solution time to the maximum solution time among all settings in a given figure.

In Figure 6a, the average time ratios for instances in classes C20, C40 and C80 are reported. Observe that `GLPK` outperforms other solvers even commercial solvers such as `CPLEX` or `Gurobi`. Observe too that `SCIP` is around 4 times slower than any other solver. Figure 6b and Figure 6c show the average time ratios for instances in classes C160 and C320, respectively. Note that `GLPK` is not included in these figures since the solver went out of memory when solving such large instances. Observe that `SCIP` performs still the worst for class C160 but it is competitive for Class C320. However, `Gurobi` seems to be the best solver for solving large instances overall. In the previous section, we mentioned that C++ implementation is up to 20% faster than `OOESCplex` for class C320. However, from figure 6c, we observe that `CPLEX` is around 15% faster than `OOESCplex`. So, `CPLEX` should be competitive with the C++ implementation.



(a) Instances C20, C40 and C80 (b) Instances C160 (c) Instances C320

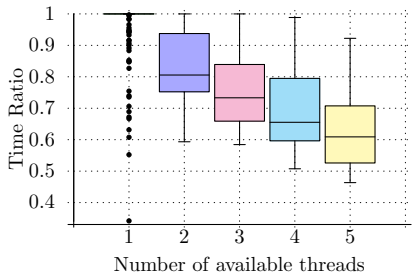
Figure 6: The performance of using different solvers

5.3. Parallelization

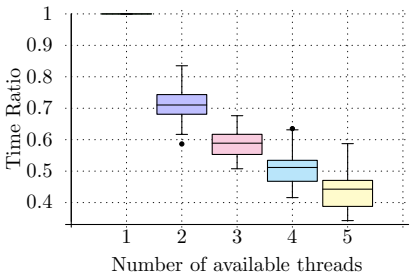
In this section, we compare the performance of `OOESCplex` when multiple threads are available under different parallelization techniques, i.e., *PriorityQueue*, *Horizontal*, *Vertical*, and Diagonal. Moreover, in this section, only medium and large instances, i.e., those in classes C80, C160 and C320, are used.

Figure 7 shows a comparison between solution time ratios when $t \in \{1, 2, 3, 4, 5\}$ are employed. Specifically, in Figure 7a, the box plots of averages (of the solution time ratios) over all parallelization techniques are reported. Observe that the median of the computational time decreases almost linearly by employing more number of threads. Also, by comparing the medians of the box plot corresponding to 1 and 5 threads, it is evident that the improvement percentage is around 40%.

In Section 4, we mentioned that *PriorityQueue* maximizes the utilization of the available threads. This can be observed from Figure 7b in which it illustrates the box plot of the solution time ratios for different threads when only *PriorityQueue* is used. Observe that the time ratio decreases significantly when more



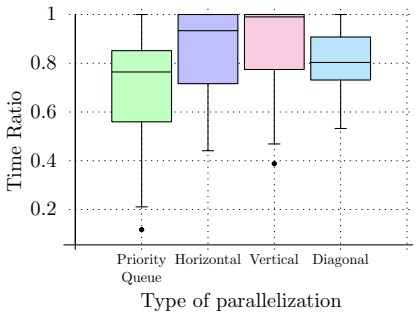
(a) Average of all parallelization techniques



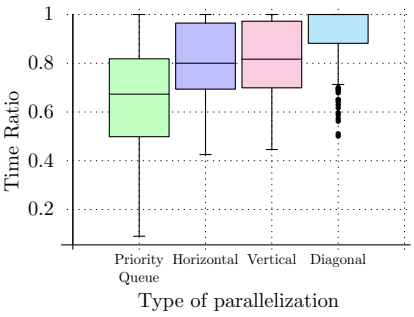
(b) For only *PriorityQueue*

Figure 7: Performance of `OOESCplex` when using multiple threads

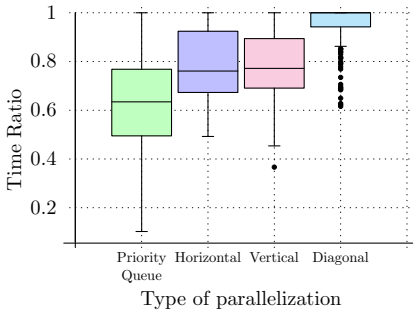
threads are employed. In fact, by comparing the medians of the box plot corresponding to 1 and 5 threads, it is evident that the improvement percentage is around 55%. To highlight the effect of *PriorityQueue* even further, in Figure 8, we compare the solution time ratios under different parallelization techniques when $t \in \{2, 3, 4, 5\}$ are available. Observe that *PriorityQueue* performs the best and the percentage decrease of the median is between 25% and 40%. This is mainly because in other parallelization techniques, the solution time depends directly on the most difficult rectangle generated after splitting the nondominated frontier. So, the usage of threads can be more unbalanced compared to *PriorityQueue*.



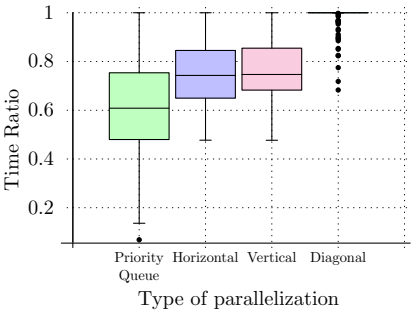
(a) 2 threads available



(b) 3 threads available



(c) 4 threads available



(d) 5 threads available

Figure 8: Performance of `OOESCplex` when using multiple threads under different parallelization techniques

To show that our numerical results are not limited to just`OOESCplex`, we conducted a set of experiments with `SCIP`. Figure 9 shows the time ratios (for all instances) for `SCIP` when multiple threads are available and *PriorityQueue* is employed. Observe that the time ratio decreases significantly when more threads are available. In fact, by comparing the medians of the box plot corresponding to 1 and 5 threads, it is evident that the improvement percentage is around 50%.

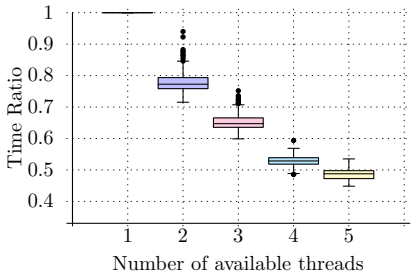


Figure 9: The Performance of `SCIP` when employing *PriorityQueue*

6. Conclusion

We developed `OOES.jl`, and open-source package for optimizing a linear function over the set of efficient solutions for BOMILPs in julia. `OOES.jl` supports execution on multiple processors and exploits different parallelization techniques. It was numerically shown that parallelization helps to improve the solution time significantly. Another desirable characteristic of the package is that it allows users to employ different commercial and non-commercial solvers for solving single-objective optimization problems arising during the course of the algorithm. The computational study showed that even non-commercial solvers can perform quite well. Finally, it was numerically shown that a julia package can be competitive with a C++ package.

References

[1] Abbas, M., Chaabane, D., 2006. Optimizing a linear function over an integer efficient set. European Journal of Operational Research 174 (2), 1140–1161.

[2] Benson, H. P., 1993. A bisection-extreme point search algorithm for optimizing over the efficient set in the linear dependence case. Journal of Global Optimization 3 (1), 95–111.

[3] Boland, N., Charkhgard, H., Savelsbergh, M., 2017. A new method for optimizing a linear function over

the efficient set of a multiobjective integer program. European Journal of Operational Research 260 (3), 904 – 919.

[4] Chaabane, D., Brahmi, B., Ramdani, Z., 2012. The augmented weighted tchebychev norm for optimizing a linear function over an integer efficient set of a multicriteria linear program. International Transactions in Operational Research 19 (4), 531–545.

[5] Djamal, C., Marc, P., 2010. A method for optimizing over the integer efficient set. Journal of industrial and management optimization 6 (4), 811.

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

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

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

[9] Pettersson, W., Ozlen, M., 2017. A parallel approach to bi-objective integer programming. ANZIAM Journal 58, 69–81.

[10] Sierra-Altamiranda, A., Charkhgard, H., 2017. A new exact algorithm to optimize a linear function over the set of efficient solutions for bi-objective mixed integer linear programming.`https://goo.gl/CSXHBc`. Last accessed: March 21, 2018.

[11] Yamamoto, Y., 2002. Optimization over the efficient set: overview. Journal of Global Optimization 22 (1-4), 285.

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