**Submitted to**

**Vol. 00, No. 0, Xxxxx 0000, pp. 000–000**

**ISSN 0000-0000, EISSN 0000-0000**



# MultiObjectiveAlgorithms.jl: a Julia package for solving multi-objective optimization problems

Oscar Dowson
Dowson Farms, Auckland, New Zealand, oscar@dowsonfarms.co.nz

Xavier Gandibleux
Nantes Université, Nantes, France, xavier.gandibleux@univ-nantes.fr

Gökhan Kof
Graduate School of Sciences and Engineering, Koç University, Istanbul, Türkiye

**Abstract.** We present `MultiObjectiveAlgorithms.jl`, an open-source Julia library for solving multi-objective optimization problems written in JuMP. `MultiObjectiveAlgorithms.jl` implements ten different solution algorithms which all rely on an iterative scalarization of the problem from a multi-objective optimization problem to a sequence of single-objective subproblems. As part of this work, we extended JuMP to support vector-valued objective functions. Because it is based on JuMP, `MultiObjectiveAlgorithms.jl` can use a wide variety of commercial and open-source solvers to solve the single-objective subproblems, and it supports problem classes including linear, integer, conic, semi-definite, and general nonlinear.

**Key words:** optimization; multi-objective; Julia; JuMP

## 1. Introduction

This paper presents `MultiObjectiveAlgorithms.jl` (`MOA.jl`), an open-source Julia (Bezanson et al. 2017) library for solving multiple objective optimization models written in JuMP (Lubin et al. 2023). Multi-objective optimization programs are well-studied in the literature; see, for example, Ehrgott and Gandibleux (2000), Ehrgott (2005), Emmerich and Deutz (2018), Ehrgott et al. (2026) and the references therein. We formulate a multi-objective optimization problem as:

$$\begin{aligned} \min_{\mathbf{x}\in\mathbb{R}^n} \;& f_0(\mathbf{x}) \\ \text{s.t.} \;& f_i(\mathbf{x}) \in S_i, \quad i = 1, \ldots, m, \end{aligned} \tag{1}$$

where $f_0 : \mathbb{R}^n \to \mathbb{R}^{p_0}$ where $p_0 \geq 2$, and $f_i : \mathbb{R}^n \to \mathbb{R}^{p_i}$ where $p_i \geq 1$, and $S_i \subset \mathbb{R}^{p_i}$ for $i = 1, \ldots, m$. The functions $f$ and sets $S$ abstract over the various classes of mathematical optimization problems, from mixed-integer linear to semi-definite programs. For more details on this standard form, see Legat et al. (2021).

### 1.1. Solutions

Let $\mathcal{X}$ denote the set of all feasible solutions of Problem (1) and let $\mathcal{Y} = \{f_0(\mathbf{x}) : \mathbf{x} \in \mathcal{X}\}$ denote the set of corresponding objective vectors. The set $\mathcal{X}$ belongs to *decision-space*, and the set $\mathcal{Y}$ belongs to *objective-space*.

Unlike a single-objective program, where an optimal solution is reported via a single vector for $\mathbf{x}$, a multi-objective program replaces the notion of *optimality* with the notion of *efficiency* (Ehrgott 2005). Let $\leq$ be defined for $\mathbf{a}, \mathbf{b} \in \mathbb{R}^{p_0}$ such that $\mathbf{a} \leq \mathbf{b} \iff \mathbf{b} - \mathbf{a} \in \mathbb{R}^{p_0}_{+}$ and $\mathbf{a} \neq \mathbf{b}$. Consider $\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{X}$. If $f_0(\mathbf{x}_1) \leq f_0(\mathbf{x}_2)$, we say $\mathbf{x}_1$ *dominates* $\mathbf{x}_2$ and $f_0(\mathbf{x}_1)$ dominates $f_0(\mathbf{x}_2)$. A feasible solution $\mathbf{x} \in \mathcal{X}$ is an *efficient* solution if there are no feasible solutions that dominate it. The corresponding objective vector, $\mathbf{y} = f_0(\mathbf{x})$, is called a *nondominated point*. A complete set of efficient solutions is denoted $\mathcal{X}_E \subseteq \mathcal{X}$, and the set of all nondominated points $\mathcal{Y}_N = \{f_0(\mathbf{x}) : \mathbf{x} \in \mathcal{X}_E\}$ is called the *nondominated set*. A visualization of decision- and objective-space is provided in Figure 1.

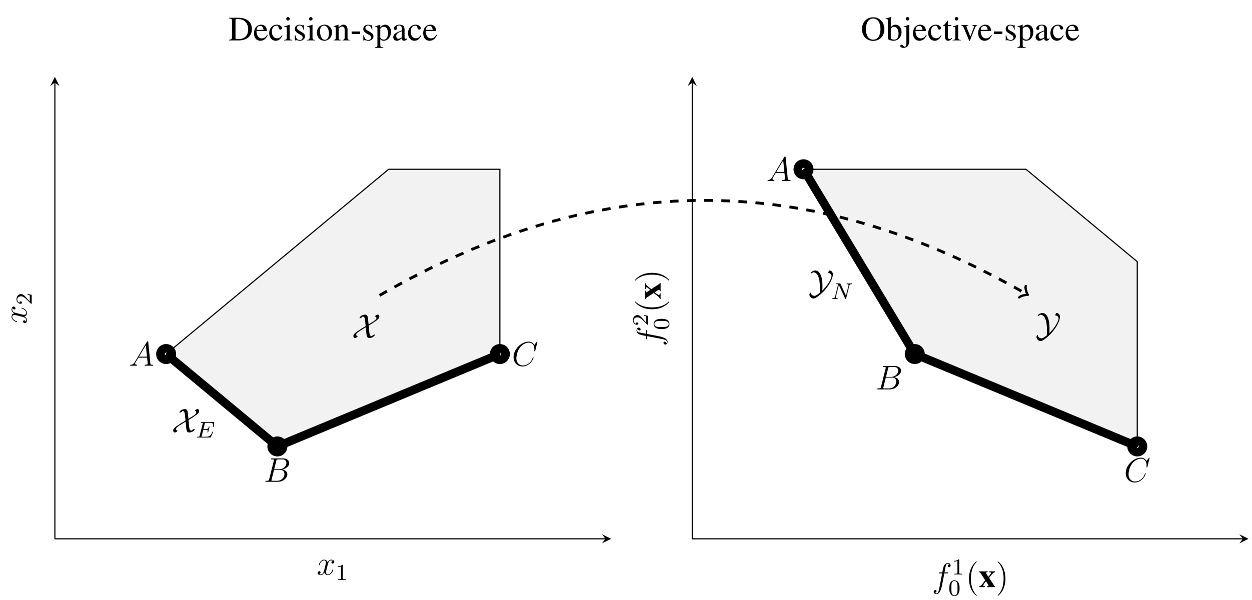


**Figure 1** **Visualization of decision- and objective-space for a linear program with two decision variables and two objectives to minimize. The feasible set $\mathcal{X}$ in decision-space is projected to the set $\mathcal{Y}$ in objective-space via the objective function. The solid line A-B-C in decision-space is a complete set of efficient solutions $\mathcal{X}_E$. The solid line A-B-C in objective-space is the corresponding non-dominated set $\mathcal{Y}_N$.**

Due to the existence of equivalent solutions, that is, solutions $\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{X}_E$ such that $f_0(\mathbf{x}_1) = f_0(\mathbf{x}_2)$, the definition of $\mathcal{X}_E$ is refined: the *maximum complete set* $\mathcal{X}_{\overline{E}}$ contains all solutions $\mathbf{x}$ corresponding to all non-dominated points $\mathbf{y} = f_0(\mathbf{x})$ and may contain multiple $\mathbf{x}$ for the same $\mathbf{y}$; a *complete set* $\mathcal{X}_E$ contains at least one $\mathbf{x}$ for each non-dominated $\mathbf{y}$; and a *minimum complete set* $\mathcal{X}_{\underline{E}}$ contains exactly one $\mathbf{x}$ for each $\mathbf{y}$. Each of these sets may be infinite, for example, in the case of a multi-objective linear program in which the non-dominated set is piecewise linear. For this paper, we are concerned only with the *minimum complete set* $\mathcal{X}_{\underline{E}}$.

Some algorithms for solving Problem (1) may be further restricted to finding only a *representative set* of efficient solutions $\mathcal{X}_R \subseteq \mathcal{X}_{\underline{E}}$. A *singleton representative set* is a special case when $|\mathcal{X}_R| = 1$. Another special case concerns the notion of a *supported* non-dominated point. A non-dominated point $\mathbf{y}$ is *supported* if $\exists \mathbf{w} > 0$ for which the corresponding efficient solution $\mathbf{x}$ is an optimal solution of $\min_{\mathbf{x}\in\mathcal{X}} \mathbf{w}^\top f_0(\mathbf{x})$, otherwise it is *non-supported*. Some algorithms compute only a *minimum supported set* of efficient solutions $\mathcal{X}_{\underline{SE}} \subseteq \mathcal{X}_{\underline{E}}$, which is the subset of a *minimum complete set* such that all corresponding nondominated points are supported. The set of solutions computed by an algorithm also depends on the problem type. For example, some algorithms may compute a minimum supported set of efficient solutions for continuous linear programs, but a representative set of efficient solutions for integer or nonlinear programs. Other algorithms are applicable only for particular problem classes, for example, they may require that all decision variables are discrete.

### 1.2. Solution algorithms

A large number of solution algorithms have been proposed for Problem (1). These algorithms can be divided into four groups along two axes according the nature of solutions returned: algorithms computing *exact* or *approximated* solutions, and algorithms *with* or *without* exploitation of preference information from the decision maker. `MOA.jl` implements *exact* algorithms *without* preference exploitation based on mathematical programming.

At a high level, the algorithms in `MOA.jl` solve a sequence of subproblems that are variations of the following scalar-objective problem:

$$\begin{aligned} SP(\mathbf{u},\mathbf{v}) = \min_{\mathbf{x}\in\mathbb{R}^n} \; & \mathbf{u}^\top f_0(\mathbf{x}) \\ \text{s.t. } & f_i(\mathbf{x}) \in S_i, \quad i = 1, \ldots, m \\ & f_0(\mathbf{x}) \leq \mathbf{v}, \end{aligned} \tag{2}$$

where $\mathbf{u} \in \mathbb{R}^{p_0}$ is a *weighted-sum* vector that combines the vector-valued objective into a scalar objective, and the vector $\mathbf{v} \in \mathbb{R}^{p_0}$ places an upper bound constraint on the objective values. By iteratively changing **u** and **v**, algorithms can find a minimum complete set. Some algorithms may add additional variables and constraints on **x**.

`MOA.jl` is a meta-solver. In high-level Julia code it modifies the vectors **u** and **v**, and it uses third-party solvers to solve the resulting single-objective subproblems. Because `MOA.jl` is built on JuMP, it can use a wide variety of commercial and open-source solvers to solve the single-objective subproblems, and it supports problem classes including linear, integer, conic, semi-definite, and general nonlinear. The solutions that `MOA.jl` returns, while subject to various theoretical guarantees, depend on the chosen algorithm and problem structure.

As an example, consider the epsilon-constraint algorithm (Haimes et al. 1971). Pseudo-code for solving the bi-objective case ($p_0 = 2$) of Problem (1) is provided in Algorithm 1. The function $SP$ means solve Problem (2) and return the optimal objective value. The function `filter_dominated` means remove all elements of the set $\mathcal{S}$ that are dominated by another element. The algorithm is controlled by a scalar parameter $\varepsilon > 0$. When the objective functions can take only discrete integer values and $\varepsilon = 1$, the algorithm is guaranteed to find a minimum complete set, otherwise, the solution is a representative set.

**Algorithm 1:** Pseudo-code for the bi-objective epsilon-constraint algorithm.

$opt_1^* \leftarrow \mathrm{SP}([1,0],[\infty,\infty])$

$opt_2^* \leftarrow \mathrm{SP}([0,1],[\infty,\infty])$

$\mathcal{S} \leftarrow \{\},\ ub_1 \leftarrow opt_1^*,\ ub_2 \leftarrow \infty$

**while** $ub_2 > opt_2^*$ **do**

  $ub_2 \leftarrow \mathrm{SP}([0,1],[ub_1,\infty])$ and store optimal $\mathbf{x}^*$

  $ub_1 \leftarrow ub_1 + \varepsilon$

  $\mathcal{S} \leftarrow \mathcal{S} \cup \{(\mathbf{x}^*, f_0(\mathbf{x}^*))\}$

**end**

**return** `filter_dominated`$(\mathcal{S})$

### 1.3. Contribution and outline

The main contribution of this paper is the Julia library `MultiObjectiveAlgorithms.jl`, which we abbreviate to `MOA.jl`. A novel feature of `MOA.jl` is its modular ability to add new solution algorithms. This makes it easy for researchers to develop new solution algorithms and compare them against the existing state of the art. To enable `MOA.jl`, we extended JuMP to support vector-valued objective functions. The ease to which users can now model, solve, and analyze multi-objective optimization problems is a significant contribution of our work.

The purpose of this paper is not to provide a mathematical description of the various algorithms that are implemented in `MOA.jl`. Instead, our contribution is to make a variety of existing algorithms available in a high quality open-source library. We refer readers interested in the algorithmic details to the extant literature (see Table 1) and, more practically, to the code archive at Dowson et al. (2026).

As a secondary contribution, we explain our design goals and provide a comparison to alternative software. Our hope is that our positive experiences developing `MOA.jl` provides motivation and inspiration for developers to add similar features in other algebraic modeling languages.

The rest of this paper is organized as follows. In Section 2 we briefly introduce `MOA.jl` by way of a code example. In Section 3 we provide some brief numerical benchmarks. In Section 4 we provide a comparison of `MOA.jl` to related work. Section 5 presents our design principles, before we conclude in Section 6.

## 2. Code example

As a didactic example of the usage of `MOA.jl`, we consider a knapsack problem with two linear objectives, one constraint, and binary variables:

$$\begin{aligned} \max_{\mathbf{x}\in\mathbb{R}^n} \quad & C\mathbf{x} \\ \text{s.t.} \quad & \mathbf{w}^\top\mathbf{x} \le b \\ & x_i \in \{0,1\}, \quad i = 1,\dots,n, \end{aligned} \tag{3}$$

where $C$ is a 2-by-$n$ matrix of costs, **w** is a vector of length $n$ for the weight of each item, and $b$ is the capacity of the knapsack. The code to implement this problem in JuMP is:

```
using JuMP, HiGHS
import MultiObjectiveAlgorithms as MOA
function solve_knapsack(C::Matrix, w::Vector, b::Real, algorithm)
    @assert size(C) == (2, length(w))
    model = Model(() -> MOA.Optimizer(HiGHS.Optimizer))
    set_silent(model)
    set_attribute(model, MOA.Algorithm(), algorithm)
    @variable(model, x[1:length(w)], Bin)
    @objective(model, Max, C * x)
    @constraint(model, w' * x <= b)
```

```
11      optimize!(model)
12      assert_is_solved_and_feasible(model)
13      return [value.(x; result = i) => objective_value(model; result = i)
14              for i in 1:result_count(model)]
15  end
```

Lines 1 and 2 import the required Julia packages. Line 3 defines a Julia function with the relevant arguments (the corresponding `end` is at line 15). Line 4 performs a sanity check of the input data. Line 5 creates a new JuMP model, with `MOA.jl` as the optimizer. Because `MOA.Optimizer` is a meta-solver, it requires a mathematical programming solver as input. Here we use the HiGHS solver (Huangfu and Hall 2018), which is available as `HiGHS.Optimizer` in the `HiGHS.jl` Julia package. Line 6 turns off printing. Line 7 controls which algorithm `MOA.jl` uses to solve the problem; Section 2.1 provides a complete list, but examples are `MOA.EpsilonConstraint()` and `MOA.Dichotomy()`. Lines 8–12 define and solve the JuMP model using typical syntax. Note how the objective function on Line 9 is a vector. Lines 13–14 use JuMP's support for returning multiple solutions to iterate over the number of available results and return a vector of $(\mathbf{x}, \mathbf{y})$ pairs corresponding to an efficient solution and its non-dominated point.

The `solve_knapsack` function can be called as follows:

```
julia> C, w, b = rand(1:20, 2, 10), rand(1:20, 10), 50.0

julia> solve_knapsack(C, w, b, MOA.EpsilonConstraint())
4-element Vector{Pair{Vector{Float64}, Vector{Float64}}}:
 [1.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0] => [58.0, 64.0]
 [1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0] => [65.0, 59.0]
 [1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0] => [69.0, 56.0]
 [1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0] => [71.0, 42.0]

julia> solve_knapsack(C, w, b, MOA.Lexicographic())
2-element Vector{Pair{Vector{Float64}, Vector{Float64}}}:
 [1.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0] => [58.0, 64.0]
 [1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0] => [71.0, 42.0]
```

Note how the size of the solution set is algorithm-dependent. Moreover, for all algorithms, the solution returned by `MOA.jl` is a finite set of points. Thus, no algorithm can return a complete set of efficient solutions if variables are continuous. The decision to return only a finite set of points is a significant trade-off that we make for practical reasons. For more detail on this decision, see Section 5.2.

### 2.1. Supported algorithms

Table 1 summarizes the algorithms supported by `MOA.jl`. There are currently ten algorithms implemented. Some algorithms, such as `Chalmet`, are specialized for a class of bi-objective problems. Others, such as `DominguezRios` support higher-dimensional MIP problems. The `Hierarchical` algorithm is identical to the algorithms implemented in solvers such as HiGHS (Huangfu and Hall 2018). We based our implementation on the description in Gurobi Optimization, LLC (2025b). The `RandomWeighting` algorithm is a iterative sampling algorithm for finding a representative set. We include it in `MOA.jl` because it appears in GAMS (GAMS Software GmbH 2025).

| Algorithm | Citation | Restriction | Lines | Solution |
|---|---|---|---|---|
| `Chalmet` | Chalmet et al. (1986) | $p_0 = 2$ | 112 | Minimum complete set |
| `Dichotomy` | Aneja and Nair (1979) | $p_0 = 2$ | 95 | Minimum supported set |
| `DominguezRios` | Dominguez-Rios et al. (2021) | IP | 220 | Minimum complete set |
| `EpsilonConstraint` | Haimes et al. (1971) | $p_0 = 2$ | 101 | Minimum complete set |
| `Hierarchical` | Gurobi Optimization, LLC (2025b) | | 92 | Singleton representative set |
| `KirlikSayin` | Kirlik and Sayın (2014) | IP | 141 | Minimum complete set |
| `Lexicographic` | Isermann (1982) | | 111 | Representative set |
| `RandomWeighting` | GAMS Software GmbH (2025) | | 61 | Representative set |
| `Sandwiching` | Koenen et al. (2023) | | 128 | Minimum supported set |
| `TambyVanderpooten` | Tamby and Vanderpooten (2021) | IP | 231 | Minimum complete set |

**Table 1 Summary of the algorithms supported by `MOA.jl`. The `Restriction` column indicates if the algorithm is restricted to a particular problem class: $p_0 = 2$ means the algorithm is restricted to bi-objective problems, IP means the algorithm is restricted to pure integer linear subproblems. *Lines* is the number of lines of Julia code required to implement each algorithm. The *Solution* columns assumes the problem is pure linear or pure integer; for other problem classes the set of solutions may be different.**

Each algorithm is relatively simple to implement, requiring less than 250 lines of Julia code. One reason so few lines are required is that we rely heavily on JuMP to manage the complexity of interacting with the single-objective subproblems.

Aside from the supported problem dimension, the biggest differences between the algorithms is the type of solutions that they return. Some algorithms, such as `TambyVanderpooten` can return points from a *minimum complete set* for discrete problems. Other algorithms, such as `Hierarchical` return only a single point, and `Lexicographic` returns a representative set of points corresponding to all lexicographic permutations of the objective vector (this can be disabled to return only a singleton representative set for high-dimensional problems).

## 3. Benchmarks

In this section we give a brief set of benchmarks to illustrate how `MOA.jl` can be used with different algorithms and subproblem solvers. We use the multi-objective binary knapsack instances from Tamby and Vanderpooten (2021). The formulation is the same as Problem (3), except that the objective matrix $C$ is $p$-by-$n$ instead of 2-by-$n$. We consider three problem sizes: $(p,n)=(2,200)$, $(p,n)=(3,100)$, and $(p,n)=(4,50)$. For each problem size we use the 10 problem instances from Tamby and Vanderpooten (2021). We used Julia v1.12.5 (Bezanson et al. 2017), JuMP v1.30.0 (Lubin et al. 2023), and `MOA.jl` v1.11.0. For the subproblem solvers, we use Gurobi, v13.0 (Gurobi Optimization, LLC 2025a) and HiGHS, v1.12 (Huangfu and Hall 2018), and we use the `Dichotomy`, `EpsilonConstraint`, and `TambyVanderpooten` algorithms. When $p > 2$ we study only `TambyVanderpooten`. For each case, we report the mean over the 10 instances of: the number of non-dominated solutions found; the number of single-objective subproblems solved; the total runtime in seconds; and the percentage of time spent inside the subproblem solver. All computations were carried out on a 2020 Apple M1 Mac Mini with 16 GB of RAM. We summarize the benchmark results in Table 2. The code and data to reproduce our experiments is available at Dowson et al. (2026).

| $p$ | $n$ | Algorithm | Solver | Solutions | Subproblems | Time [s] | Time in subproblem [%] |
|---|---|---|---|---|---|---|---|
| 2 | 200 | `Dichotomy` | Gurobi | 34.7 | 68.9 | 1 | 88 |
| 2 | 200 | `Dichotomy` | HiGHS | 34.9 | 68.1 | 6 | 98 |
| 2 | 200 | `EpsilonConstraint` | Gurobi | 405.4 | 813.8 | 29 | 99 |
| 2 | 200 | `EpsilonConstraint` | HiGHS | 405.4 | 813.8 | 351 | 100 |
| 2 | 200 | `TambyVanderpooten` | Gurobi | 405.4 | 818.2 | 31 | 99 |
| 2 | 200 | `TambyVanderpooten` | HiGHS | 405.4 | 818.6 | 366 | 100 |
| 3 | 100 | `TambyVanderpooten` | Gurobi | 4,090.0 | 14,724.4 | 673 | 97 |
| 4 | 50 | `TambyVanderpooten` | Gurobi | 2,240.1 | 20,570.2 | 1,042 | 68 |

**Table 2 Benchmark results for the multiobjective knapsack problems of Tamby and Vanderpooten (2021) where: "Solutions" is the size of the non-dominated set $|Y_{\mathcal{N}}|$; "Subproblems" is the number of single-objective subproblems that were solved during the algorithm; "Time [s]" is the total solve time in seconds; and "Time in subproblem [%]" is the percentage of the total solve time that was spent in the solve routine of the subproblems. All values are the mean across the 10 problem instances for each problem size.**

We verified that our solutions match those reported in Tamby and Vanderpooten (2021); they report the average number of non-dominated solutions as $405.4$ for $(p, n) = (2, 200)$, $4090.0$ for $(p, n) = (3, 100)$, and $2, 240.1$ for $(p, n) = (4, 50)$, which matches the results for our exact algorithms. Since the `Dichotomy` algorithm finds only a minimum *supported* set, it does not find all non-dominated solutions. Interestingly, our solve times are 2–5 times slower than those reported in Tamby and Vanderpooten (2021), even though we solve a similar number of subproblems. Moreover, the "Time in subproblem [%]" column shows that the majority of the total solve time is dominated by the time spent solving the subproblems. This means that Julia, JuMP, and `MOA.jl` are not bottlenecks in the solution process. Thus, we attribute the runtime differences to either the machine and solver used by Tamby and Vanderpooten (2021), or by solver-specific options that they do not report (we used the default settings).

For a given problem size and algorithm, we observe differences in the number of subproblems solved by Gurobi and HiGHS (and also between the iteration counts reported by Tamby and Vanderpooten (2021)). Moreover, HiGHS finds an additional non-dominated solution in instances $9$ and $10$ of the $(p, n) = (2, 200)$ case with the `Dichotomy` algorithm. This additional solution is a valid non-dominated point, but it is not *supported*. In theory, the algorithm should not find this point. However, HiGHS finds the non-supported point because the scalarized objective value is within the MIP gap of a nearby *supported* non-dominated point. Thus, the different number of subproblems explored by each solver is attributable to the different sequences of iterations that can occur when there are multiple near-optimal primal solutions for a particular subproblem.

One surprising result in Table 2 is the order-of-magnitude difference in runtime between HiGHS and Gurobi. Working with the HiGHS developers, we analyzed the instances in more detail. We verified that the performance difference was purely an artifact of the solver, and that it was not related to Julia, JuMP, `MOA.jl`, or how subproblems are efficiently cached and modified between solves. Instead, the reason for the discrepancy is simple: the subproblems are trivial mixed-integer programs with one constraint and 200 (or fewer) decision variables. HiGHS does not attempt to detect and exploit this problem structure; instead it runs generic techniques that work well on large-scale problems, but are relatively expensive on these small-scale problems.

## 4. Comparison to related works

Table 3 compares `MOA.jl` to related mathematical programming software for modeling and solving multi-objective optimization problems. We compare four modeling language approaches, `MOA.jl`, AMPL (AMPL Optimization Inc. 2025), MOCVXPY (Diamond and Boyd 2016, Salomon et al. 2025), and GAMS (GAMS Software GmbH 2025), and five open-source solver approaches, BenSOLVE (Löhne and Weißing 2017), HiGHS (Huangfu and Hall 2018), inner (Csirmaz 2021), PaMILO (Bökler et al. 2023), and PolySCIP (Borndörfer et al. 2016).

Many commercial solvers (for example, COPT, CPLEX, Gurobi, Hexaly, and Xpress) have support for multiple objectives. We choose HiGHS as a single representative example of this type of solver because they all implement similar variations of the `Lexicographic` and `Hierarchical` algorithms. None support returning a set of multiple efficient solutions. To the best of our knowledge, no other popular algebraic modeling languages (for example, Pyomo (Bynum et al. 2021)) support multi-objective problems. MATLAB has a variety of support for multiobjective optimization, including the `paretosearch` function for returning a representative set of linear and nonlinear optimization problems (MATLAB 2025). We omit it from Table 3 because we could not confirm what algorithms it implements or what solvers it uses.

Considering the large and well-developed body of literature that exists around multi-objective problems, mathematical programming based software packages for solving such problems are sparse and relatively feature poor. Where support does exist, most packages focus on returning a single representative solution that is some mix of a lexicographic and hierarchical blending.

The lack of general purpose libraries means that practitioners who want to solve a multi-objective problem and return multiple solutions typically code their own implementation. In our opinion, the lack of a general purpose library for solving multi-objective optimization problems has limited the applicability of multi-objective optimization because it requires practitioners to have a reasonable level of software development proficiency. As a secondary effect, practitioners may implement only a single (and simple) solution algorithm, instead of experimenting with a range of more sophisticated solution algorithms.

The lack of general purpose libraries also hurts *developers* of novel solution algorithms. In most cases, papers that develop new general purpose algorithms publish reproducible source code, but the code is often problem-specific and not intended for general purpose use. If future authors implement their algorithm in `MOA.jl`, they immediately gain the ability to model and solve problems using JuMP, and their work is also available to the wider community. We see this point as a positive, if minor, contribution of our work.

## 5. Design principles

We based the design of `MOA.jl` on the following design principles.

| | *Modeling-language-based* | | | | *Open-Source Solver-based* | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | `MOA.jl` | AMPL | MOCVXPY | GAMS | BenSOLVE | HiGHS | inner | PaMILO | PolySCIP |
| Programming Language | Julia | Custom | Python | Custom | C | Many | C | C | C |
| License | BSD-3 | Commercial | Apache | Commercial | GPL-2 | MIT | GPL-3 | Custom | Custom |
| Modeling Language | JuMP | AMPL | CVXPY | GAMS | | | | | Zimpl |
| Solvers | Many | Many | Many | Many | GLPK | HiGHS | GLPK | CPLEX/Gurobi | SCIP |
| Multiple Solutions | Yes | | Yes | Yes | Yes | | Yes | Yes | Yes |
| *Problem class* | | | | | | | | | |
| Linear | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| Integer | Yes | Yes | | Yes | | Yes | | Yes | Yes |
| Nonlinear | Yes | Yes | Convex | Yes | | | | Quadratic | |
| *Algorithms* | | | | | | | | | |
| Benson | | | Yes | | Yes | | Yes | | Yes |
| Bökler | | | | | | | | Yes | |
| Chalmet | Yes | | | | | | | | |
| Dichotomy | Yes | | | | | | | | |
| DominguezRios | Yes | | | | | | | | |
| EpsilonConstraint | Yes | | | Yes | | | | | |
| Hierarchical | Yes | Yes | | | | Yes | | | |
| KirlikSayin | Yes | | | | | | | | |
| Lexicographic | Yes | Yes | | Yes | | Yes | | | |
| RandomWeighting | Yes | | | Yes | | | | | |
| Sandwiching | Yes | | | Yes | | | | | |
| TambyVanderpooten | Yes | | | | | | | | |

**Table 3 General comparison of mathematical programming software packages for multi-objective optimization. The *Algorithms* are based on the naming scheme in Table 1. The exceptions are `Benson` (Benson 1998), and `Bökler` (Bökler and Mutzel 2015). The algorithms implemented by BenSOLVE, inner, MOCVXPY, and PolySCIP are different, but they are all variants of Benson's. PaMILO and PolySCIP have custom open-source licenses that prevent commercial use.**

### 5.1. Modular algorithms

The last decade has led to proliferation of novel solution algorithms for multi-objective optimization problems. We point to the work of Dominguez-Rios et al. (2021) and Tamby and Vanderpooten (2021) as examples. Because algorithm design and comparison is an on-going research challenge, we wanted a design that makes writing new algorithms easy. As a secondary consideration, we wanted to make switching between algorithms simple to enable speed and solution quality comparisons between algorithms.

We achieve these goals by leveraging Julia's strength for multiple dispatch. Each algorithm subtypes the `MOA.AbstractAlgorithm` type and implements a single function, `MOA.minimize_multiobjective`. The return value is a termination status and a list of solution points (if they exist). For example, a new algorithm could be implemented as:

```
import MultiObjectiveAlgorithms as MOA
struct NewAlgorithm <: MOA.AbstractAlgorithm end
function MOA.minimize_multiobjective(model::MOA.Optimizer, ::NewAlgorithm)
    solutions = MOA.SolutionPoint[]
    # Implement the algorithm using `model.f::MOI.AbstractVectorFunction`
    if error
        return MOI.OTHER_ERROR, nothing
    end
    return MOI.OPTIMAL, solutions
end
```

This algorithm can then be immediately used from JuMP with:

```
set_attribute(model, MOA.Algorithm(), NewAlgorithm())
```

Behind the scenes, `MOA.jl` will build the problem, convert from maximization to minimization (if necessary), and convert the returned vector of solutions into the form required by JuMP. The separation of algorithm and modeling interface simplifies writing an algorithm; each of the algorithms implemented in `MOA.jl` is less than 250 lines of code.

### 5.2. A solution is a finite set of points

Unlike the single-objective optimization literature, the multi-objective algorithm literature has not converged on a single definition of what constitutes a "solution." As described in Section 1.1, there are at least three common definitions: the *maximum complete set*, a *complete set*, and a *minimum complete set*. These sets may be finite and discrete or continuous and infinite.

In `MOA.jl`, we choose to ignore this complexity. We do not require that an algorithm returns a minimum complete set. We define a solution as a finite set of vectors $\mathbf{x}$ with corresponding objective vectors $\mathbf{y} = f_0(\mathbf{x})$. We order solutions lexicographically based on the objective vector and the objective sense; the first solution is best. We make no quality assertions about the returned list, other than that each objective vector is non-dominated. We make this decision for pragmatic reasons: it makes the library simpler to implement, and it provides most users with most of what they want in practice, which is the ability to find a representative set of diverse solutions to their problem. The trade-off is that it requires users to have some theoretical knowledge to choose an appropriate solution algorithm for their problem type, and to understand whether the returned solution is a representative set or a minimum complete set.

### 5.3. A single objective sense and a vector-valued objective

When designing `MOA.jl`, we had robust discussions about two alternate approaches: is a multi-objective optimization problem defined by a set of independent scalar objectives, each of which may have a different objective sense, or is it a vector-valued function with a single objective sense? Many users would prefer the former, since it allows them to, for example, minimize one objective and maximize another. The latter requires them to normalize all scalar objectives into the same objective sense by multiplying maximization objectives by $-1$.

Because `MOA.jl` is based on JuMP, which itself relies on the MathOptInterface standard form (Legat et al. 2021), we chose to implement the library as a single objective sense and a vector-valued objective function. Although more restrictive from a user-perspective, this significantly simplified the implementation of `MOA.jl`, and we have not found this decision to be a burden in practice. Note that our decision is purely an artifact of our decision to build on JuMP; GAMS and MOCVXPY, for example, allow a different objective sense for each scalar objective.

## 6. Conclusions

This paper has presented `MultiObjectiveAlgorithms.jl` (`MOA.jl`), an open-source Julia package for solving multi-objective optimization problems written in JuMP. Readers are directed to Dowson et al. (2026) for the archived source code and to `https://github.com/jump-dev/MultiObjectiveAlgorithms.jl` for the latest version.

`MOA.jl` implements ten different solution algorithms, which all rely on an iterative scalarization of the problem from a multi-objective optimization problem to a sequence of single-objective subproblems. Each algorithm is relatively simple to implement, requiring less than 250 lines of code, and the modular design of the library makes it easy to add new algorithms. Because it is based on JuMP, `MOA.jl` can use a wide variety of commercial and open-source solvers to solve the single-objective subproblems, and it supports problem classes ranging from linear, to conic, semi-definite, and general nonlinear.

The only gap between recent advances in algorithms for multi-objective optimization and the current design of the library is our decision to return a finite set of points as the solution. As future work, we would like to extend `MOA.jl` to support solutions such as nondominated facets and the associated open and closed points. The correct implementation of this is an open question, particularly with how to represent these solutions in JuMP.

To conclude, we hope that our positive experience developing `MOA.jl` serves as motivation and design inspiration for others to add similar features to algebraic modeling languages that do not currently support multi-objective optimization problems.

## Acknowledgments

`MOA.jl` is the successor to the `vOptSolver` project (Gandibleux et al. 2017). The development of vOptSolver started in the ANR/DFG-14-CE35-0034-01 research project vOpt (2015–2019).

## References

AMPL Optimization Inc (2025) Multiple objective. URL `https://mp.ampl.com/modeling-mo.html`, date visited: 2025-06-21.

Aneja Y, Nair K (1979) Bicriteria transportation problem. *Management Science* 25(1):73–78.

Benson H (1998) An outer approximation algorithm for generating all efficient extreme points in the outcome set of a multiple objective linear programming problem. *Journal of Global Optimization* 13:1–24.

Bezanson J, Edelman A, Karpinski S, Shah VB (2017) Julia: A Fresh Approach to Numerical Computing. *SIAM Review* 59(1):65–98.

Bökler F, Mutzel P (2015) Output-sensitive algorithms for enumerating the extreme nondominated points of multi-objective combinatorial optimization problems. Bansal N, Finocchi I, eds., *Algorithms - ESA 2015*, 288–299 (Berlin, Heidelberg: Springer Berlin Heidelberg).

Bökler F, Nemesch L, Wagner MH (2023) PaMILO: A solver for multi-objective mixed integer linear optimization and beyond. *arXiv preprint arXiv:2207.09155* URL `https://arxiv.org/abs/2207.09155`.

Borndörfer R, Schenker S, Skutella M, Strunk T (2016) PolySCIP. Greuel GM, Koch T, Paule P, Sommese A, eds., *Proceedings of the International Conference on Mathematical Software*, volume 9725 of *Lecture Notes in Computer Science*.

Bynum ML, Hackebeil GA, Hart WE, Laird CD, Nicholson BL, Siirola JD, Watson JP, Woodruff DL (2021) *Pyomo–optimization modeling in Python*, volume 67 (Cham, Switzerland: Springer Science & Business Media), third edition.

Chalmet L, Lemonidis L, Elzinga D (1986) An algorithm for the bi-criterion integer programming problem. *European Journal of Operational Research* 25(2):292–300.

Csirmaz L (2021) Inner approximation algorithm for solving linear multiobjective optimization problems. *Optimization* 70(7):1487–1511.

Diamond S, Boyd S (2016) CVXPY: A Python-embedded modeling language for convex optimization. *The Journal of Machine Learning Research* 17(1):2909–2913.

Dominguez-Rios M, Chicano F, Alba E (2021) Effective anytime algorithm for multiobjective combinatorial optimization problems. *Information Sciences* 565(7):210–228.

Dowson O, Gandibleux X, Kof G (2026) MultiObjectiveAlgorithms.jl: a Julia package for solving multi-objective optimization problems. URL `http://dx.doi.org/10.1287/ijoc.2025.1449.cd`, available for download at https://github.com/INFORMSJoC/2025.1449.

Ehrgott M (2005) *Multicriteria Optimization* (Berlin; New York: Springer), 2nd edition, ISBN 978-3-540-21398-7.

Ehrgott M, Gandibleux X (2000) A survey and annotated bibliography of multiobjective combinatorial optimization. *OR Spektrum* 22(4):425–460.

Ehrgott M, Köksalan M, Kadziński M, Deb K (2026) Fifty years of multi-objective optimization and decision-making: From mathematical programming to evolutionary computation. *European Journal of Operational Research* 330(1):1–25.

Emmerich M, Deutz A (2018) A tutorial on multiobjective optimization: fundamentals and evolutionary methods. *Natural Computing* 17:585–609.

GAMS Software GmbH (2025) Multi-objective optimization. URL `https://www.gams.com/50/docs/T_LIBINCLUDE_MOO.html`, date visited: 2025-05-21.

Gandibleux X, Soleilhac G, Przybylski A, Ruzika S (2017) vOptSolver: an open source software environment for multiobjective mathematical optimization. IFORS2017: 21st Conference of the International Federation of Operational Research Societies. July 17-21, 2017. Quebec City (Canada).

Gurobi Optimization, LLC (2025a) Gurobi Optimizer. URL `https://www.gurobi.com`, date visited: 2025-05-21.

Gurobi Optimization, LLC (2025b) Multiple objectives. URL `https://docs.gurobi.com/projects/optimizer/en/current/features/multiobjective.html`, date visited: 2025-05-21.

Haimes Y, Lisdon L, Wismer D (1971) On a bicriterion formulation of the problems of integrated system identification and system optimization. *IEEE transactions on systems, man, and cybernetics* 3:296–297.

Huangfu Q, Hall JAJ (2018) Parallelizing the dual revised simplex method. *Mathematical Programming Computation* 10(1):119–142.

Isermann H (1982) Linear lexicographic optimization. *OR Spectrum* 4(4):223–228.

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

Koenen M, Balvert M, Fleuren H (2023) A renewed take on weighted sum in sandwich algorithms: Modification of the criterion space. Technical report, CentER, Center for Economic Research, Tilburg, Netherlands, URL `https://research.tilburguniversity.edu/en/publications/795b6c0c-c7bc-4ced-9d6b-a6d9409866be`, date visited: 2026-04-08.

Legat B, Dowson O, Dias Garcia J, Lubin M (2021) MathOptInterface: a data structure for mathematical optimization problems. *INFORMS Journal on Computing* 34(2):672–689.

Lubin M, Dowson O, Garcia JD, Huchette J, Legat B, Vielma JP (2023) JuMP 1.0: Recent improvements to a modeling language for mathematical optimization. *Mathematical Programming Computation* 15(3):581–589.

Löhne A, Weißing B (2017) The vector linear program solver bensolve – notes on theoretical background. *European Journal of Operational Research* 260(3):807–813.

MATLAB (2025) paretosearch – find points in pareto set. URL `https://au.mathworks.com/help/gads/paretosearch.html`, date visited: 2025-06-29.

Salomon L, Dörfler D, Löhne A (2025) MOCVXPY: a CVXPY extension for multiobjective optimization. *arXiv preprint arXiv:2510.21010* URL `https://arxiv.org/abs/2510.21010`.

Tamby S, Vanderpooten D (2021) Enumeration of the nondominated set of multiobjective discrete optimization problems. *INFORMS Journal on Computing* 33(1):72–85.