`OOESAlgorithm.jl`: A comprehensive julia package to optimize a linear function over the efficient set of Biobjective Mixed Integer Linear Programming problems.

Alvaro Sierra-Altamiranda & Hadi Charkhgard

USER MANUAL

Contents

Description

The package described in this document is a user-friendly open-source julia package (see LICENSE) that contains a criterion space search algorithm for optimizing a linear function over the set of efficient solutions of bi-objective mixed integer linear programming problems. This is a julia v1.0.2 project written in Linux (Ubuntu). This package can be found at the following link:

`https://github.com/alvsierra286/OOESAlgorithm.jl`,

while a detailed explanation of how the algorithm works can be found at:

Sierra-Altamiranda, A. and Charkhgard, H., A New Exact Algorithm to Optimize a Linear Function Over the Set of Efficient Solutions for Bi-objective Mixed Integer Linear Programming, to appear at the INFORMS Journal On Computing.

Characteristics

Some important characteristics of this package are:

- **The package can solve any (both structured and unstructured) biobjective mixed integer linear problem. The following problem classes are supported:**
  1. Objectives: 2 linear objectives and one extra objective to be optimized over the efficient set.
  2. Constraints: 0 or more linear (both inequality and equality) constraints
  3. Variables:
     - Binary variables
     - Integer variables
     - Continuous variables
- **A biobjective mixed integer linear instance can be provided as input to the package in 3 ways:**
  1. `JuMP` Model
  2. LP file format
  3. MPS file format
- **Any mixed integer linear programming solver supported by MathProgBase.jl can be used.**
  - `OOESAlgorithm.jl` automatically installs `GLPK.jl` as the default single-objective mixed integer linear programming solver. If the user desires to use any other MIP solver, it must be separately installed. Note that `OOESAlgorithm.jl` has been successfully tested with:
    1. GLPK - v4.61
    2. SCIP - v5.0.1

    * The file /.julia/packages/SCIP/.../src/mpb_interface.jl of the package SCIP.jl v0.6.1 has to be modified manually. In lines 185, 186, 192, 193, 207 and 208, numvar(m) has to be changed by MathProgBase.numvar(m).
  3. Gurobi - v7.5
  4. CPLEX - v12.7

- **Supports parallelization.**
  - Can use how many threads are available when julia is opened.
  - Can implement different parallelization types.
- **Can be easily modified to compute the entire Pareto-optimal frontier.**
  - Can compute the nondominated frontier by just tuning a parameter.

Installation

Next you will find the instruuctions to install `OOESAlgorithm.jl`.

Installation Guide

`OOESAlgorithm.jl` is a package for Julia. To use `OOESAlgorithm.jl`, first download and install Julia. Note that the whole ecosystem has been tested using Julia v1.0.2 and hence we can cannot guarantee whether it will work with previous versions of Julia. Thus, it is important that a Julia version v1.0.2 or higher is properly installed and available on your system. Our package is available at the `METADATA.jl` repository, therefore, from Julia, the latest released version of `OOESAlgorithm.jl` is installed by using the package environment:

```
(v1.0) pkg> add OOESAlgorithm
```

You may also install the package with the latest updates using the built-in package manager:

```
julia> import Pkg
julia> Pkg.clone("https://github.com/alvsierra286/OOESAlgorithm.jl")
julia> Pkg.build("OOESAlgorithm")
```

Dependencies

The dependencies of `OOESAlgorithm.jl` include the following Julia packages:

- `MathOptInterface.jl`

- `MathProgBase.jl`
- `JuMP.jl`
- `GLPKMathProgInterface.jl`
- `GLPK.jl`

Note that `GLPK.jl` is installed with the package, and therefore, used as the default single-objective mixed integer programming (MIP) solver. However, our package relies on the MIP solvers supported by the package `MathProgBase.jl`. In this sense (if available) we strongly recommend to use other optimization solvers.

Supported Solvers

Next you will find a list of MIP solvers supported by `OOESAlgorithm.jl`.

Solver	Julia Package	License
Cbc	`Cbc.jl`	Eclipse Public License
CPLEX	`CPLEX.jl`	Commercial
FICO Xpress	`Xpress.jl`	Commercial
GLPK	`GLPK.jl`	General Public License
Gurobi	`Gurobi.jl`	Commercial
MOSEK	`MathProgBaseMosek.jl`	Commercial
SCIP	`SCIP.jl`	ZIB Academic License

Observe that some of the supported solvers are available for public and academic use. We refer the users to the links in the table of supported solvers to follow guidelines for correct installation. A solvers' benchmark is available at *Sierra-Altamiranda, A. and Charkhgard, H. (2018). OOES.jl: A julia package for optimizing a linear function over the set of efficient solutions for bi-objective mixed integer linear programming.*

Getting Started

This quick start guide will introduce the main concepts of `OOESAlgorithm.jl`.

First Steps

Once `OOESAlgorithm.jl` is installed, to use `OOESAlgorithm.jl` in your programs, you just need to say:

```
julia> using OOESAlgorithm
```

The main function exported when using `OOESAlgorithm.jl` is called OOES, and returns the optimal solution for the problem of optimizing a linear function over the set of efficient solutions for biobjective mixed integer linear programs. To use this function, we need to provide the input in file format (LP, MPS) or a `JuMP` model. For example:

LP format:

```
julia> OOES("input_file.lp")
```

MPS format:

```
julia> OOES("input_file.mps")
```

`JuMP` model:

```
julia> model = JuMP.Model()
after creating the model:
julia> OOES(model)
```

`OOESAlgorithm.jl` assumes that all the functions are in minimization form. In further section we explain how to consider problems in maximization form. `OOESAlgorithm.jl` automatically creates an output summary file with the value of the objective functions and a solution file with the value of the variables. We can also assign the optimal solution to a variable:

```
julia> optimal_solution = OOES(model)
```

The solution returned has the following information:

```
vector with the values of the objective functions:
julia> optimal_solution.obj_vals

vector with the values of the variables:
julia> optimal_solution.vars

true when solution is optimal over a nondominated point, false otherwise:
julia> optimal_solution.fxopt
```

Creating The Input For `OOESAlgorithm.jl`

Consider the following BOMILP problem:

$$\begin{aligned} \text{min } & 3x_1 + x_2 \\ \text{min } & -x_1 - 2x_2 \\ \text{subject to: } & x_2 \leq 3 \\ & 3x_1 - x_2 \leq 6 \\ & x1 \in \mathbb{R}_+,\ x_2 \in \mathbb{Z}_+, \end{aligned}$$

and let the linear function we want to optimize over the set of efficient solutions of the problem be:

$$-2x_1 + 3x_2.$$

To represent this problem in any format, we first consider a generic minimization function equivalent to the sum of all the variables. The order of the variables in this objective function will correspond to the order of the variables in the output. Then we add the constraints of the problem, and finally we add the 2 objective functions and the function to be optimized over the efficient set as additional equality constraints with RHS equal to 0. The first two functions will be the objective functions of the BOMILP program and the third function will be the linear function to be optimized over the efficient set. In some cases, when using a `JuMP` model, objective functions can be confused with some other equality constraints with RHS equal to 0. In this case, replace the equality constraint with two constraints (one $\geq$ and the other $\leq$).
The following box represents the text of the BOMILP problem in LP file format:

```
Minimize
obj: x1+x2
Subject To
c1: x2 <= 3
c2: 3 x1 - x2 <= 6
c3: 3 x1 + x2 = 0
c4: - x1 - 2 x2 = 0
c5: -2 x1 + 3 x2 = 0
Bounds
0 <= x1 <= 1e10
0 <= x2 <= 1e10
General
x2
End
```

The following box represents the text of the BOMILP problem in MPS file format:

```
ROWS
N obj
L c1
L c2
E c3
E c4
E c5
COLUMNS
x1    obj    1
x1    c2     3
x1    c3     3
x1    c4    -1
x1    c5    -2
MARK0000   'MARKER'    'INTORG'
x2              obj           1
x2              c1            1
x2              c2           -1
x2              c3            1
x2              c4           -2
x2              c5            3
MARK0001   'MARKER'    'INTEND'
RHS
rhs             c1            3
rhs             c2            6
BOUNDS
UP bnd x1 10000000000
UP bnd x2 10000000000
ENDATA
```

And finally, the following box shows the julia commands to represent the BOMILP as a `JuMP` model:

```
julia> using JuMP
julia> model = JuMP.Model()
julia> @variable(model, 0 <= x1 <= 1e10)
julia> @variable(model, 0 <= x2 <= 1e10, Int)
julia> @objective(model, Min, x1 + x2)
julia> @constraint(model, x2 <= 3)
julia> @constraint(model, 3x1 - x2 <= 6)
julia> @constraint(model, 3x1 + x2 == 0)
julia> @constraint(model, -x1 - 2x2 == 0)
julia> @constraint(model, -2x1 + 3x2 == 0)
```

Advanced Features

In this section we explain some of the advanced features available in `OOESAlgorithm.jl`. The advanced features are divided in tuning parameters, solver selection and parallelization. Additionally, we include some test files that run our package under some scenarios with the default parameters. For instance, lets assume that our input is a JuMP.Model() called model.

Tuning Parameters

Next we explain some basic parameters with their default values, and how can they be modified.

Time Limit

Time limit is a termination condition of `OOESAlgorithm.jl`. When the running time of `OOESAlgorithm.jl` reach the time limit, it automatically stops. By default, `OOESAlgorithm.jl` finish the current iteration before exiting. The default value for the time limit is 86400.0 seconds, which is equivalent to one day. The following command must be followed in order to modify the time limit:

```
julia> OOES(model, timelimit=α)
```
where α is a floating number with the new time limit in seconds.

For example, if we want to run our package for one hour, then we must type:

```
julia> OOES(model, timelimit=3600.0)
```

Relative Gap

The relative gap is a termination condition of the single-objective MIP solver implemented in `OOESAlgorithm.jl`. The relative gap by default is 1.0^{-6}. To change this value, we must type:

```
julia> OOES(model, relative_gap=β)
```
where β is a floating number with the new relative gap.

For example, if we want to run our package with a relative gap of 1.0^{-9}, then we must type:

```
julia> OOES(model, relative_gap=1.0e-9)
```

Optimization Sense

The opotimization sense is a parameter that let us declare if the objective function is in minimization or maximization form. By default, the sense of all objective functions is minimization, however, we can change that by tuning the vector sense in `OOESAlgorithm.jl`. For example, if our first objective function is in minimization form, while the second and third functions are in maximization form, we must type:

```
julia> OOES(model, sense=[:Min, :Max, :Max])
```

Solver Selection

`OOESAlgorithm.jl` offers two different options to select the single-objective MIP solver to implement. One of the options is by tuning the parameter mipsolver, and the other by tuning the parameter mip_solver.

mipsolver

`OOESAlgorithm.jl` automatically detects if some of the most popular solvers are installed, and mipsolver represents one of the solvers with an integer number. The list of the solvers with their respective mipsolver number is shown next:

Solver	mipsolver
`GLPK.jl`	1
`Gurobi.jl`	2
`CPLEX.jl`	3
`SCIP.jl`	4
`Xpress.jl`	5

For example, if we want to run our package with `CPLEX.jl` (if CPLEX is available and properly installed), then we must type:

```
julia> OOES(model, mipsolver=3)
```

mip_solver

We can tune the parameter mip_solver when using `OOESAlgorithm.jl` to explore advanced settings in the single-objective MIP solvers. This feature is recommended for advanced users. A list with some of the solvers with their respective mip_solver code is shown next:

Solver	mip_solver
`Gurobi.jl`	GurobiSolver(OutputFlag=ϕ, Threads=ρ, MIPGap=γ)
`CPLEX.jl`	CplexSolver(CPX_PARAM_SCRIND=ϕ, CPX_PARAM_THREADS=ρ, CPX_PARAM_EPGAP=γ)
`SCIP.jl`	SCIPSolver("display/verblevel", ϕ, "limits/gap", γ)
`Xpress.jl`	Xpress.XpressSolver(XPRS_OUTPUTLOG=ϕ, XPRS_THREADS=ρ, XPRS_STOP_MIPGAP=γ)

Where ϕ indicates the level of verbosity of the MIP solver, ρ is the number of threads (if multiple processors are available, we strongly recommend to declare $\rho = 1$ and exploit parallelization by tuning the parallelization parameters), and γ is the relative gap. For example, if we want to run our package with `Gurobi.jl` (Given that Gurobi is available and properly installed), with no screen information, 1 thread and relative gap of 1.0^{-12} then we must type:

```
julia> OOES(model, mip_solver=GurobiSolver(OutputFlag=0, Threads=1, MIPGap=1e-12))
```

Note that, if the solver is not detected by the package, it must be added manually as shown in the next example:

```
julia> using OOESAlgorithm, MathProgBaseMosek
```

Parallelization

In this section, we explain two important parameters to exploit parallelization in our package. In order to enable multiple processors, we must type in the terminal the following code:

```
~ $ julia -p Ω
where Ω is the number of processors to enable.
```

For example, if we want to enable 3 processors, then we must type:

```
~ $ julia -p 3
```

Number Of Threads

Once we enable multiple processors, we can tune a parameter to declare the number of threads we want to use. This parameter should have a value lesser or equal to the number of available processors. The following command must be followed in order to modify the number of threads:

```
julia> OOES(model, threads=σ)
where σ is an integer number with the new number of threads to use.
```

For example, if we want to run our package with three threads, then we must type:

```
julia> OOES(model, threads=3)
```

Parallelization Techniques

We can tune a parameter to exploit different parallelization techniques. There are a total of 4 techniques, three based on cuts in the criterion space and one based on the elements of the priority queue of the algorithm. Further details on how the parallelization techniques work can be found at:
Sierra-Altamiranda, A. and Charkhgard, H. (2018). OOES.jl: A julia package for optimizing a linear function over the set of efficient solutions for bi-objective mixed integer linear programming.
To choose between the different parallelization techniques, we tune the parameter parallelization, which is an integer number. The list parallelization techniques with their respective parameter value are shown next: The following command must be followed in order to choose a different parallelization technique:

Technique	parallelization
Horizontal cuts	1
Vertical cuts	2
Diagonal cuts	3
Elements of the Priority Queue	4

```
julia> OOES(model, parallelization=θ)
```
where θ is an integer number that represents the parallelization technique to use.

For example, if we want to run our package, using diagonal cuts, then we must type:

```
julia> OOES(model, parallelization=3)
```

Computing The Pareto-Optimal Frontier

The package can easily be modified to compute the entire Pareto-optimal frontier of a biobjective mixed integer linear programming. The name of the parameter to tune is pareto_frontier, a boolean variable that indicates that the package will compute the Pareto-optimal frontier when the value is true. For example, if we want to get the Pareto-optimal frontier, using our package, we must type:

```
julia> OOES(model, pareto_frontier=true)
```

The algorithm implemented to compute the Pareto-optimal frontier is the *Triangle Splitting Method*. The output of the package contains the points in the Pareto-optimal frontier, organized in non-decreasing order of the first objective function, and a boolean parameter that indicates if such point is connected to the next point in the frontier by a line or not. If connected, then the connection is a line contained in the Pareto-optimal frontier.
Please be aware of the following observation: to be sure that our package is computing the exact nondominated frontier, the input file must have 3 objectives just as if we were optimizing the linear function over the efficient set, i.e., 3 constraints equal to 0. The order of the objectives is the same as the explained in the subsection 'Creating The Input For `OOESAlgorithm.jl`'. The third objective function will automatically be obviated by the package, therefore we may consider any simple function. Additionally, the package returns a vector with the points Pareto-optimal frontier and the solutions correspondent to such points. An example to obtain the vector is shown next:

```
julia> Pareto_optimal_frontier = OOES(model, pareto_frontier=true)
```

In this example, the variable Pareto_optimal_frontier is a vector with as many positions as the points discovered in the nondominated frontier. Each position will contain the following information:

vector with the values of the objective functions in the point ξ of the Pareto-optimal frontier:
`julia> Pareto_optimal_frontier[ξ].obj_vals`

vector with the values of the variables in the point ξ of the Pareto-optimal frontier:
`julia> Pareto_optimal_frontier[ξ].vars`

true when the point ξ is connected to the next point of the Pareto-optimal frontier by a line:
`julia> Pareto_optimal_frontier[ξ].fxopt`

The parameter pareto_frontier also supports parallelization, however, it doesn't support the fourth technique, i.e., parallelization=4.

Testing

We provide our package with a test located in OOESAlgorithm.jl/test/ to check the installation. After installing `OOESAlgorithm.jl`, open julia in a new terminal, go to **pkg** environment and execute the following code:

```
(v1.0) pkg> test OOESAlgorithm
```

The test executes a total of three examples, one of them a JuMP model, and the other two LP and MPS files. If the installation of `OOESAlgorithm.jl` was successful and the solver returns the optimal values for the objective functions, the following message should appear:

```
Solution test 1:  [-276.667,   -257.333,   -443.0]
Test 1 Successful

Solution test 2:  [-276.667,   -257.333,   -443.0]
Test 2 Successful

Solution test 3:  [-276.667,   -257.333,   -443.0]
Test 3 Successful

    Testing OOESAlgorithm tests passed
```

Support and Citing

The software in this ecosystem was developed as part of academic research. If you would like to help support it, please star the repository as such metrics may help us secure funding in the future. If you use `OOESAlgorithm.jl` software as part of your research, teaching, or other activities, we would be grateful if you could cite:

1. *Sierra-Altamiranda, A. and Charkhgard, H., A New Exact Algorithm to Optimize a Linear Function Over the Set of Efficient Solutions for Bi-objective Mixed Integer Linear Programming*, INFORMS Journal On Computing, to appear.
2. *Sierra-Altamiranda, A. and Charkhgard, H. (2018). OOESAlgorithm.jl: A julia package for optimizing a linear function over the set of efficient solutions for bi-objective mixed integer linear programming.*

Contributions

This package is written and maintained by Alvaro Sierra-Altamiranda. Please fork and send a pull request or create a GitHub issue for bug reports or feature requests.