Wroclaw University of Science and Technology
Faculty of Fundamental Problems of Technology

RobRecSolver package. User Manual and API Reference

Report: Ser. PRE nr 3

Author:
Mikita Hradovich

2019

Wrocław
University
of Science
and Technology

Contents

1 Introduction

RobRecSolver is a package written in a Julia programming language developed to test performance of algorithms proposed in M. Hradovich, A. Kasperski, P. Zieliński *Robust recoverable 0-1 optimization problems under polyhedral uncertainty*[1]. This work was supported by the grant "Discrete optimization problems under uncertainty - models and algorithms" (2017/25/B/ST6/00486) by the National Center for Science (Narodowe Centrum Nauki).

1.1 Installing RobRecSolver

If you are familiar with Julia you can quickly install RobRecSolver and CPLEX:

```
julia> Pkg.add("CPLEX")
julia> Pkg.clone("https://github.com/nikagra/RobRecSolver.jl.git")
```

Note: You need a working installation of CPLEX Optimizer. See 2.2Getting CPLEX Optimizer for more information.

1.2 Citing

You can cite [1] by using the following BibTeX snippet:

```
@article{HKZ19,
        Author = {Mikita Hradovich and Adam Kasperski and Zieli{\'n}ski,
            Pawe{\l}},
        Title = {Robust recoverable 0-1 optimization problems under
            polyhedral uncertainty},
        Journal = {European Journal of Operational Research},
        Volume = {278},
        Number = {1},
        Pages = {136-148},
        Year = {2019},
        Doi = {10.1016/j.ejor.2019.04.017},
        Url = {https://doi.org/10.1016/j.ejor.2019.04.017}
}
```

2 Instalation Guide

This guide will briefly guide you through installing Julia, CPLEX Optimizer and RobRecSolver with all of its dependencies.

2.1 Getting Julia

Version of Julia programming language required by JuMP and consequently by RobRecSolver is `0.6`. You can build Julia from source or use the binaries.

Download links and more detailed instructions are available on the Julia website.

2.2 Getting CPLEX Optimizer

RobRecSolver package depends on CPLEX.jl which in turn requires a working installation of CPLEX Optimizer with a license, which is free for faculty members and graduate teaching assistants.

CPLEX Optimizer must be downloaded and installed separately. Check CPLEX.jl for further instructions.

2.3 Getting RobRecSolver

RobRecSolver package is *not* yet registered in the METADATA.jl repository. To install it, use `Pkg.clone` command:

```
julia> Pkg.clone("https://github.com/nikagra/RobRecSolver.jl.git")
```

Since RobRecSolver contains REQUIRE file, that file will be used to determine which registered packages RobRecSolver depends on, and they will be automatically installed.

2.4 Updating RobRecSolver

In order to update package run the following sequence of commands (`;` symbol at the start of the Julia's REPL enters shell mode):

```
julia> cd(Pkg.dir("RobRecSolver"))
julia> ;
shell> git fetch --all --tags --prune && git checkout tags/<version>
julia> Pkg.resolve()
```

You may want to check Pro Git written by Scott Chacon and Ben Straub to learn more about git version control system.

3 Manual

3.1 Getting Started

Package consists of a number of functions implementing algorithms described in [1] as well as some utility functions. The easiest way to experiment with them is to use package in Julia's interactive session (or REPL which stands for read-eval-print loop). For example:

```
$ julia
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: https://docs.julialang.org
   _ _   _| |_  __ _   |  Type "?help" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.6.3 (2018-05-28 20:20 UTC)
 _/ |\__'_|_|_|\__'_|  |  Official http://julialang.org/ release
|__/                   |  x86_64-w64-mingw32

julia> 1 + 1
2
```

Interactive session may be useful while prototyping programs since it outputs result of each evaluation and allows to check intermediate results.

To leave interactive session enter `exit()` command or press `Ctrl+D`.

Alternatively you can evaluate source file, which uses `.jl` filename extension by convention:

```
$ julia main.jl arg
```

In the example above Julia passes argument `arg` to the program stored in source file named `main.jl` and then execute commands stored in it in non-interactive mode. Arguments passed to program are available within script in a global constant `ARGS`, thus name of the source file in global constant `PROGRAM_FILE`.

See Julia Scripting for more information about writing scripts in Julia.

To start using RobRecSolver library first import it, some of its submodules or functions with `using` keyword, then call function you are interested in, i.e. `incrementalProblem`:

```
julia> using RobRecSolver.Experiments
julia> runExperiments([100, 400, 1000], [10, 25, 100])
```

Check Julia Modules for more detailed information about using modules in Julia.

3.2 Additional Types and Functions

There is a number of types and helper functions defined to facilitate implementation of algorithms described in [1]. `ProblemDescriptor` is one of such a types. It serves as a basic interface, defining size of the problem or whether it has equal cardinality property among other properties. There two subtypes of `ProblemDescriptor`, namely `KnapsackProblemDescriptor` and `AssignmentProblemDescriptor` for each problem discussed in [1]:

```
using RobRecSolver

n = 5
knapsackProblemDescriptor = KnapsackProblemDescriptor(n)
property = hasEqualCardinalityProperty(knapsackProblemDescriptor)
println("KnapsackProblemDescriptor.hasEqualCardinalityProperty: $property")

assignmentProblemDescriptor = AssignmentProblemDescriptor(n)
property = hasEqualCardinalityProperty(assignmentProblemDescriptor)
println("AssignmentProblemDescriptor.hasEqualCardinalityProperty: $property"
    )
println("AssignmentProblemDescriptor.getCardinality: $(getCardinality(
    assignmentProblemDescriptor))")
```

Upon executing example above you will get the following output:

```
KnapsackProblemDescriptor.hasEqualCardinalityProperty: false
AssignmentProblemDescriptor.hasEqualCardinalityProperty: true
AssignmentProblemDescriptor.getCardinality: 5
```

Function `initialScenario` is another example of a helper function. It searches for good heuristic initial scenario in order to speed up some computations. Its behavior is described in the section 5.1 *Adversarial lower bound* of [1]. See example below on how to use this function:

```
using RobRecSolver

c = [2, 3]
d = [8, 9]
Γ = 10
s = initialScenario(c, d, Γ)
println("Initial scenario is ", s)
```

In this example `c` is a vector of second stage costs, `d` is vector of maximal deviations of the costs from their nominal values and Γ is a budget or the amount of uncertainty, which can be allocated to the second stage costs.

Upon running the script above you will receive the following output:

```
Initial scenario is [7.50073, 7.50073]
```

The functions `loadProperties` and `getProperty` allow to customize package parameters like solver time limits or logging for different algorithms.

Function `loadProperties` loads properties stored in an INI file from the specified file location. To change default location set `ROBRECSOLVER_CONFIG` environment variable either in Julia REPL or in `/.julia/config/startup.jl` and then reload RobRecSolver package:

```
julia> ENV["ROBRECSOLVER_CONFIG"] = "<path_to_file>"
julia> Pkg.reload("RobRecSolver")
```

Use default properties file `Pkg.dir("RobRecSolver")/conf/config.ini` as a reference. Below is an extract from it:

```
; Problem properties
[main]
lagrangianLowerBound.cplexLogging=0
lagrangianLowerBound.epsilon=0.000001
lagrangianLowerBound.overallTimeLimit=1800
lagrangianLowerBound.subproblemTimeLimit=600
```

See Appendix A.1 for full contents.

In order to reset changes simply delete environment variable and reload RobRecSolver package.

Function `getProperty` returns value for key from previously loaded properties file:

```
using RobRecSolver

ϵ = getProperty("evaluationProblem.epsilon", parameterType = Float64)
timeLimit = getProperty("evaluationProblem.timeLimit")
```

In the example above value for property `evaluationProblem.epsilon` of type `Float64` is stored in variable ϵ. Then value for property `evaluationProblem.timeLimit` of type `Int` (default) is stored in variable `timeLimit`. If properties section is not specified, section called `main` is used by default.

4 Problems

4.1 Incremental and Recoverable Problems

Section 4 *Solving the problems by MIP formulations* of [1] presents MIP formulations for incremental and recoverable problems for element exclusion neighborhood as well its simplified versions for equal cardinality problem.

Both versions of incremental problems are solved by `incrementalProblem` function. Here is example of solving incremental problem for minimum knapsack problem:

```
julia> using RobRecSolver
julia> n = 3
julia> α = 0.5
julia> c = [1, 2, 3]
julia> x = [0, 1, 1]
julia> w = [1, 2, 2]
julia> W = 3
julia> X = getKnapsackConstraints(w, W)
julia> problemDescriptor = KnapsackProblemDescriptor(n)
julia> incrementalProblem(c, α, x, X, problemDescriptor)
```

In this example we first import RobRecSolver package. Then we define a number of variables, namely `c` for a vector of nonnegative nominal second stage costs, `x` for a first stage solution, variable α for fixed number belonging to `[0,1]` as described in [1]. We also define variable `w` for a vector of item weights and `W` for knapsack capacity. Set of feasible solutions is prepared by function `getKnapsackConstraints`. It is represented as a list of anonymous functions each of which for given vector of JuMP variables returns a JuMP linear constraint. Variable `problemDescriptor` is an instance of type `KnapsackProblemDescriptor` which defines some useful properties of a problem, i.e. its size or whether it has equal cardinality property. Last step is to call function `incrementalProblem`. It will return tuple containing vector of second stage solutions and objective value.

Let us solve recoverable problem for minimum assignment problem using `recoverableProblem` function:

```
julia> using RobRecSolver
julia> m = 2
julia> α = 1.0
julia> C = [1 2; 3 1]
julia> c = [5 3; 2 4]
julia> X = getAssignmentConstraints(m)
julia> problemDescriptor = AssignmentProblemDescriptor(m)
julia> recoverableProblem(C, c, X, α, problemDescriptor)
```

As in a previous example we first define some auxiliary variables. Here `C` is a vector of nonnegative first stage costs, `C` is a vector of a nonnegative nominal second stage costs, `X` is a set of feasible solutions, α is fixed number belonging to `[0, 1]` and `problemDescriptor` is an instance of `AssignmentProblemD escriptor`. This example will return a tuple consisting of a vector of first stage solutions, a vector of second stage solutions and objective value.

4.2 Evaluation Problem

Let us take a look at a function named `evaluationProblem`, which implements **Algorithm 1** of Section 4 *Solving the problems by MIP formulations* of [1]. Here is an example of how one can use it for minimum knapsack problem:

```
julia> using RobRecSolver
julia> n = 2
julia> α = 1.0
julia> C = [4, 3]
julia> c = [2, 3]
julia> d = [8, 9]
julia> Γ = 9
julia> x = [0, 1]
julia> w = [1, 2]
julia> W = 1
julia> X = getKnapsackConstraints(w, W)
julia> problemDescriptor = KnapsackProblemDescriptor(n)
julia> evaluationProblem(C, c, d, Γ, α, x, X, problemDescriptor)
D-  2 constraints was added to this evaluation problem                Debug
     evaluation_problem.jl:1
10.0
```

We first define size of a problem `n`, parameter α, a vector of first stage costs `C`, a vector of a nonnegative nominal second stage costs `C`, a vector of maximal deviations of the costs from their nominal values `d`, budget Γ, set of feasible solutions `X` and an instance of type `KnapsackProblemDescriptor` `problemDescriptor`. We also define a vector of item weights `w` and knapsack capacity `w`. The last step is to call `evaluationProblem` passing all necessary arguments.

4.3 Lower Bounds

Section 5 *Lower bounds* of the publication contains algorithms and MIP formulations to calculate adversarial, lagrangian and cardinality selection constraint lower bounds. Corresponding functions from RobRecSolver package to calculate this lower bounds are respectively `adversarialProblem`, `lagrangianLowerBound` and `selectionLowerBound`. All of this function have very similar signatures, so as an example let us take a closer look to `adversarialProblem`. This function implements algorithm for calculating adversarial lower bound shown in the form of **Algorithm 2** of Section 5.1 *Solving the problems by MIP formulations* of the publication. Below is an example of source file solving it for minimum knapsack problem saved as `adv.jl`:

```
using RobRecSolver

n = 2
α = 0.5

w = [1, 2]
W = 1
X = getKnapsackConstraints(w, W)

C = [1, 3]
c = [3, 1]
d = [2, 2]
Γ = 2

problemDescriptor = KnapsackProblemDescriptor(n)
result = adversarialProblem(C, c, d, Γ, X, α, problemDescriptor)
println("Adversarial lower bound is ", result)
```

Assuming the above code is saved as `adv.jl`, running the above program will return the following output:

```
$ julia adv.jl
D-  3 constraints was added to this adversarial problem              Debug
    adversarial_problem.jl:1
Adversarial lower bound is 5.0
```

In this program we first define size of a problem `n`, parameter α, a vector of first stage costs `C`, a vector of a second stage costs `C`, a vector of maximal deviations of the costs from their nominal values `d`, budget Γ, set of feasible solutions `X` and an instance of type ‘KnapsackProblemDescriptor‘ ‘problemDescriptor‘ defining problem properties. Then we call ‘adversarialProblem‘ function passing all necessary arguments and printing out result. Note, that depending on package settings it also may print some additional logs.

4.4 Experiments

RobRecSolver package also contains `Experiments` submodule which can serve as a reference on how to use core package functionality in experiments. Please remember that almost all functions in `RobRecSolver.Experiments` are highly customized to serve purposes of [1]. Never the less let us take a closer look at functions presented here.

`RobRecSolver.Experiments.runExperiments` is an entry point of experiments. It accepts list of minimum knapsack problem sizes `ns`, list of minimum assignment problem sizes `ms` and optionally list of values of parameter α called α`s` and a number of instances to be generated for each value of

α called `numberOfInstances`. By default α`s` has values $0.1, 0.2, ..., 0.9$ and `numberOfInstances` equals 5:

```
using RobRecSolver.Experiments

runExperiments([100, 400, 1000], [10, 25, 100])
```

Check [1] for more information about scope of experiments.

`RobRecSolver.Experiments.saveCsv` is a helper function developed to save experiment results as CSV files. It saves data described by `columnNames` argument using data passed in `data` argument to CSV file with name `filename`:

```
using RobRecSolver

Experiments.saveCsv("item_prices.csv", ["milk" 100; "ham" 250], ["item", "
    price"])
```

The above program will save CSV file `item_prices.csv` with the following content:

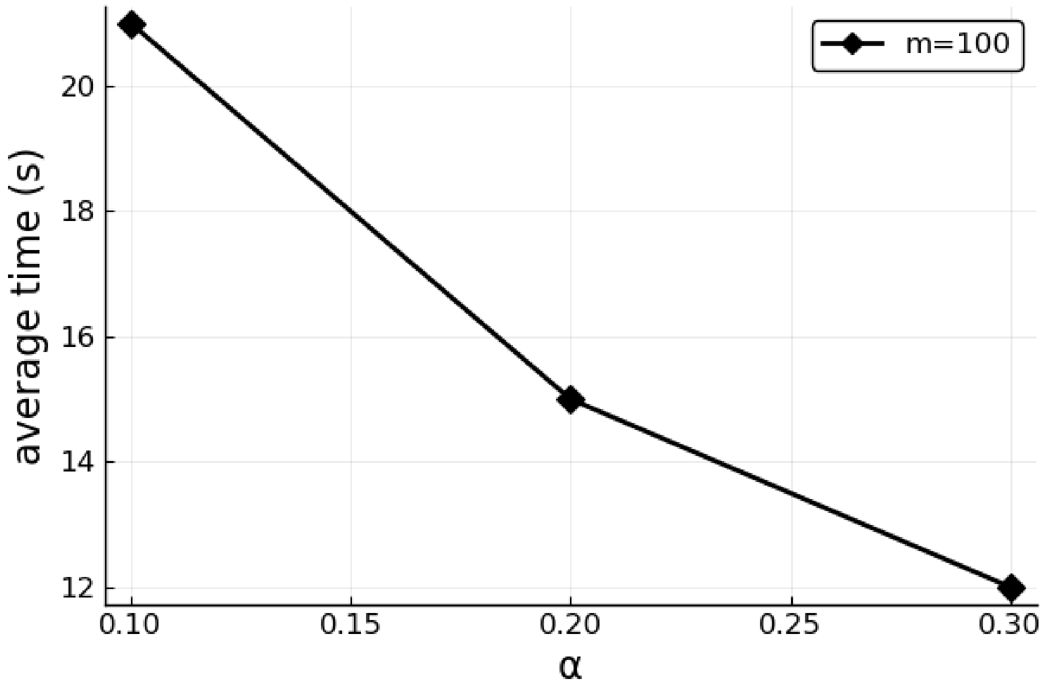
```
item,price
milk,100
ham,250
```

Function `RobRecSolver.Experiments.drawAndSavePlot` is a function used to draw plots used in [1]. It accepts a number of parameteres and is heavily based on `PyPlot` backend. As an example. the following code snippet:

```
using RobRecSolver, Plots

pyplot()
Experiments.drawAndSavePlot("plot.pdf", [0.1, 0.2, 0.3], [21, 15, 12], "α",
    "average time (s)", "m=100")
```

Will produce the following plot:



Check the Appendix A.2 for more complete reference of functions used in experiments.

5 Acknowledgements

This work was supported by the National Center for Science (Narodowe Centrum Nauki), grant 2017/25/B/ST6/00486.

A Appendices

A.1 Properties File

```
; Problem properties
[main]
lagrangianLowerBound.cplexLogging=0
lagrangianLowerBound.epsilon=0.000001
lagrangianLowerBound.overallTimeLimit=1800
lagrangianLowerBound.subproblemTimeLimit=600

selectionLowerBound.timeLimit=600
selectionLowerBound.cplexLogging=0

adversarialProblem.timeLimit=600
adversarialProblem.cplexLogging=0
adversarialProblem.epsilon=0.01

evaluationProblem.timeLimit=600
evaluationProblem.cplexLogging=0
evaluationProblem.epsilon=0.01

incrementalProblem.cplexLogging=0

minimumAssignmentProblem.cplexLogging=0

minimumKnapsackProblem.cplexLogging=0

recoverableProblem.cplexLogging=0
```

A.2 Reference

Problems

Incremental Problem

`RobRecSolver.incrementalProblem` — *Function*.

```
incrementalProblem(c, α, x, X, pd)
```

Solves incremental problem with specified costs `c`, parameter *α ∈ [0, 1]*, first stage solutions `x` and a list of constraints `X` defining a set of feasible solutions. It is subproblem of of `evaluationProblem` and `adversarialProblem`.

Check section 4 *Solving the problems by MIP formulations* of the publication for more information about this algorithm.

Arguments

- `c` : is a vector of a nonnegative nominal second stage costs.
- `α` : fixed number belonging to *[0, 1]*
- `x` : first stage solution.
- `X` : is a set of feasible solutions represented as a list functions, each of which accepts a list of JuMP variables as an argument and returns a JuMP linear constraint.
- `pd` : an instance of `ProblemDescriptor`

source

Evaluation Problem

`RobRecSolver.evaluationProblem` — *Function*.

```
evaluationProblem(C, c, d, Γ, α, x, X, pd)
```

Computes **Eval(x)** with accuracy ϵ.

Check section 4 *Solving the problems by MIP formulations* of the publication for more information about this algorithm.

Arguments

- `C` : is a vector of nonnegative first stage costs.
- `c` : is a vector of a nonnegative nominal second stage costs.
- `d` : is a vector of maximal deviations of the costs from their nominal values.
- `Γ` : is a budget, or the amount of uncertainty, which can be allocated to the second stage costs.
- `X` : is a set of feasible solutions represented as a list functions, each of which accepts a list of JuMP variables as an argument and returns a JuMP linear constraint.
- `α` : fixed number belonging to *[0, 1]*
- `pd` : an instance of `ProblemDescriptor`

source

Recoverable Problem

`RobRecSolver.recoverableProblem` — *Function*.

```
recoverableProblem(C, c, X, α, dg)
```

Solves recoverable problem **REC(c)**.

Check section 4 *Solving the problems by MIP formulations* of the publication for more information about this algorithm.

Arguments

- `C` : is a vector of nonnegative first stage costs.
- `c` : is a vector of a nonnegative nominal second stage costs.
- `X` : is a set of feasible solutions represented as a list functions, each of which accepts a list of JuMP variables as an argument and returns a JuMP linear constraint.
- `α` : fixed number belonging to *[0, 1]*
- `pd` : an instance of `ProblemDescriptor`

source

Adversarial Problem

`RobRecSolver.adversarialProblem` — *Function*.

```
adversarialProblem(C, c, d, Γ, X, α, pd)
```

Computes **Adv** with accuracy ϵ.

Check section 5.1 *Adversarial lower bound* of the publication for more information about this algorithm.

Arguments

- `C` : is a vector of nonnegative first stage costs.
- `c` : is a vector of a nonnegative nominal second stage costs.
- `d` : is a vector of maximal deviations of the costs from their nominal values.
- `Γ` : is a budget, or the amount of uncertainty, which can be allocated to the second stage costs.
- `X` : is a set of feasible solutions represented as a list functions, each of which accepts a list of JuMP variables as an argument and returns a JuMP linear constraint.
- `α` : fixed number belonging to *[0, 1]*
- `pd` : an instance of `ProblemDescriptor`

source

Selection Lower Bound

`RobRecSolver.selectionLowerBound` — *Function*.

```
selectionLowerBound(C, c, d, Γ, X, α, dg)
```

Computes selection lower bound.

Check section 5.2 *Selection lower bound* of the publication for more information about this algorithm.

Arguments

- `C` : is a vector of nonnegative first stage costs.
- `c` : is a vector of a nonnegative nominal second stage costs.
- `d` : is a vector of maximal deviations of the costs from their nominal values.
- `Γ` : is a budget, or the amount of uncertainty, which can be allocated to the second stage costs.
- `X` : is a set of feasible solutions represented as a list functions, each of which accepts a list of JuMP variables as an argument and returns a JuMP linear constraint.
- `α` : fixed number belonging to *[0, 1]*
- `pd` : an instance of `ProblemDescriptor`

source

Lagrangian Lower Bound

`RobRecSolver.lagrangianLowerBound` — *Function*.

```
lagrangian_lower_bound(C, c, d, Γ, X, l, dg)
```

Computes Lagrangian low er bound.

Check section 5.3 *Lagrangian lower bound* of the publication for more information about this algorithm.

Arguments

- `C` : is a vector of nonnegative first stage costs.
- `c` : is a vector of a nonnegative nominal second stage costs.
- `d` : is a vector of maximal deviations of the costs from their nominal values.
- `Γ` : is a budget, or the amount of uncertainty, w hich can be allocated to the second stage costs.
- `X` : is a set of feasible solutions represented as a list functions, each of w hich accepts a list of JuMP variables as an argument and returns a JuMP linear constraint.
- `l` : value of parameter *l=Γm(1-\alpha)* 7
- `pd` : an instance of `ProblemDescriptor`

source

Additional Types and Functions

`RobRecSolver.ProblemDescriptor` — *Type*.

Base type for all problem descriptors. It is expected to has the follow ing fields:

- `n` : the size of the problem.
- `saneComputationLimit` : maximum size of the problem for w hich computing results for adversarial low er bound, recoverable low er bound, selection low er bound or lagrangian low er bound makes sense.
- `equalCardinalityProperty` : specifies if problem possess equal cardinality property.
- `cardinality` : cardinality of the problem if any.

source

`RobRecSolver.KnapsackProblemDescriptor` — *Type*.

`KnapsackProblemDescriptor` is an implementation `ProblemDescriptor` for minimum knapsack problem.

source

`RobRecSolver.AssignmentProblemDescriptor` — *Type*.

`AssignmentProblemDescriptor` is an implementation `ProblemDescriptor` for minimum assignment problem.

source

`RobRecSolver.getProblemSize` — *Function*.

```
getProblemSize(pd::ProblemDescriptor)
```

Returns the size of the problem.

Arguments

- `pd` : an instance of `ProblemDescriptor`

source

`RobRecSolver.getSaneComputationLimit` — *Function*.

```
getSaneComputationLimit(pd::ProblemDescriptor)
```

Returns maximum size of the problem for which computing results for adversarial lower bound, recoverable lower bound, selection lower bound or lagrangian lower bound makes sense.

Arguments

- `pd` : an instance of `ProblemDescriptor`

source

`RobRecSolver.hasEqualCardinalityProperty` — *Function*.

```
hasEqualCardinalityProperty(pd::ProblemDescriptor)
```

Returns whether the problem possess equal cardinality property.

Arguments

- `pd` : an instance of `ProblemDescriptor`

source

`RobRecSolver.getCardinality` — *Function*.

```
getCardinality(pd::ProblemDescriptor)
```

Returns cardinality of the problem if any.

Arguments

- `pd` : an instance of `ProblemDescriptor`

source

`RobRecSolver.initialScenario` — *Function*.

```
initialScenario(c, d, Γ)
```

Returns a good initial scenario _c_0_. It is used in computation of `evaluationProblem` and `adversarialProblem`.

Check section 5.1 *Adversarial lower bound* of the publication for more information about this algorithm.

Arguments

- `c` : vector of nonnegative nominal second stage costs.
- `d` : vector of maximal deviations of the costs from their nominal values.
- `Γ` : budget, or the amount of uncertainty, which can be allocated to the second stage costs

source

`RobRecSolver.loadProperties` — *Function*.

```
loadProperties()
```

Loads properties stored in an INI file from the specified file location. To change default location set `ROBRECSOLVER_CONFIG` environment variable either in Julia REPL or in `~/.julia/config/startup.jl` and then reload RobRecSolver package:

```
julia> ENV[ROBRECSOLVER_CONFIG] = "<path_to_file>"
```

```
julia> Pkg.reload("RobRecSolver")
```

Use default properties file `Pkg.dir("RobRecSolver")/conf/config.ini` as a reference.

In order to reset changes simply delete environment variable and reload RobRecSolver package.

source

`RobRecSolver.getProperty` — *Function*.

```
getProperty(parameter[, parameterType, section])
```

Get value for key of name `parameter` of type `parameterType` from section `section` from either default properties files or the one specified with a path in `ROBRECSOLVER_CONFIG`. Argument `parameterType` defaults to `Int` and `section` defaults to `main`.

source

Experiments

Problems

`RobRecSolver.minimumKnapsackProblem` — *Function*.

```
minimumKnapsackProblem(C, w, W)
```

Solve minimum knapsack problem using vector of costs `C`, weights `w` and overall weight limit `W`.

source

`RobRecSolver.getKnapsackConstraints` — *Function*.

```
getKnapsackConstraints(w, W)
```

Return a list of constraints defining a set of feasible solutions of a minimum knapsack problem. Each constraint is function with one parameter, which is variable of a mathematical programming model.

source

`RobRecSolver.minimumAssignmentProblem` — *Function*.

```
minimumAssignmentProblem(C)
```

Solve minimum assignment problem using vector of costs `C`.

source

`RobRecSolver.getAssignmentConstraints` — *Function*.

```
getAssignmentConstraints(m)
```

Return a list of constraints defining a set of feasible solutions of a minimum assignment problem. Each constraint is function with one parameter, which is variable of a mathematical programming model.

source

Testing Framework

`RobRecSolver.Experiments` — *Module*.

`RobRecSolver.Experiments` is a module containing all of the code regarding conduction of experiments.

source

`RobRecSolver.Experiments.generateData` — *Function*.

```
generateData(problemDescriptor::ProblemDescriptor)
```

Helper function designed to generate experiment data for each problem under consideration. It returns a tuple `(C, c, d, Γ, X)` w here

1. `C` is a vector of nonnegative first stage costs
2. `c` is a vector of a nonnegative nominal second stage costs
3. `d` is a vector of maximal deviations of the costs from their nominal values
4. `Γ` is a budget, or the amount of uncertainty, w hich can be allocated to the second stage costs
5. `X` is a set of feasible solutions represented as a list functions, each of w hich accepts a list of JuMP variables as an argument and returns a JuMP linear constraint

source

`RobRecSolver.Experiments.runExperiments` — *Function*.

```
runExperiments(ns::Array{Integer}, ms::Array{Integer};  αs = collect(0.1:0.1:0.9), numberOfInstances = 5)
```

Entry point of experiments. This function runs experiments for minimum knapsack problem w ith problem size `n` specified by the list `ns` and minimum assignment problem w ith problem size `m` specified by the list `ms`. Optional argument `αs` specify a list of parameters defining neighbourhood of some solution _x_ and optional argument `numberOfInstances` specify number of problem instances to be generated for each value of `alpha`.

Examples:

```
julia> using RobRecSolver.Experiments
julia> runExperiments([100, 400, 1000], [10, 25, 100])
```

source

`RobRecSolver.Experiments.runKnapsackExperiments` — *Function*.

```
runKnapsackExperiments(ns; αs = collect(0.1:0.1:0.9), numberOfInstances = 5)
```

Runs experiments for minimum knapsack problem.

source

`RobRecSolver.Experiments.runAssignmentExperiments` — *Function*.

```
runAssignmentExperiments(ms; αs = collect(0.1:0.1:0.9), numberOfInstances = 5)
```

Runs experiments for minimum assignment problem.

source

`RobRecSolver.Experiments.exportKnapsackResults` — *Function*.

```
exportKnapsackResults(problemDescriptor, αs, results)
```

Saves results of minimum knapsack problem experiments to CSV files and as PDF plots.

Arguments

- `problemDescriptor::ProblemDescriptor` : implementation of `ProblemDescriptor` for this problem.
- `αs::Array{Integer, 1}` : list of values of α.
- `results::Array{Float64, 1}` : three-dimentional array of results, w here first dimention specify problem, second dimention specify ratios or times results, the third one contain results for each value of α.

source

`RobRecSolver.Experiments.exportAssignmentResults` — *Function*.

```
exportAssignmentResults(problemDescriptor, αs, results)
```

Saves results of minimum assignment problem experiments to CSV files and as PDF plots.

Arguments

- `problemDescriptor::ProblemDescriptor` : implementation of `ProblemDescriptor` for this problem.
- `αs::Array{Integer, 1}` : list of values of α.
- `results::Array{Float64, 1}` : three-dimentional array of results, w here first dimention specify problem, second dimention specify ratios or times results, the third one contain results for each value of α.

source

`RobRecSolver.Experiments.saveCsv` — *Function*.

```
saveCsv(filename, data, columnNames)
```

Saves `data` described by `columnNames` to CSV file w ith name `filename`.

Examples:

```
julia> using RobRecSolver
julia> Experiments.saveCsv("item_prices.csv", ["milk" 100; "ham" 250], ["item", "price"])
```

The above command w ill create file `item_prices.csv` w ith the follow ing content:

```
item,price
milk,100
ham,250
```

source

`RobRecSolver.Experiments.drawAndSavePlot` — *Function*.

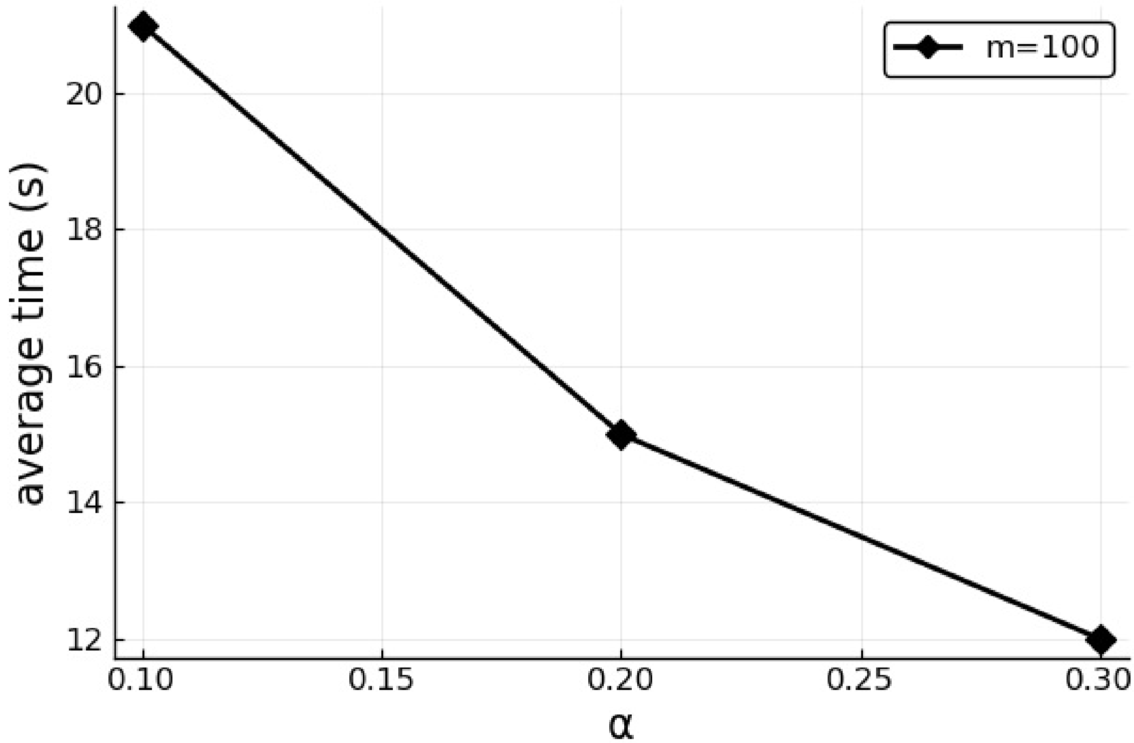
```
drawAndSavePlot(filename, x, ys, xlabel, ylabel, yslabels; linewidth=2, linestyles = [:solid :dash :dashdot :dot :solid], s
```

Draw s plot and saves it to PDF file w ith name `filename`. Here `x` is a values of 0X axis, `ys` is a columns of series, `xlabel` is label of 0X axis, `ylabel` is label of 0Y axis and `yslabels` is a labels of individual series.

Examples:

```
julia> using RobRecSolver, Plots
julia> pyplot()
julia> Experiments.drawAndSavePlot("plot.pdf", [0.1, 0.2, 0.3], [21, 15, 12], "α", "average time (s)", "m=100")
```

The above command w ill draw the plot show n below and save it as `plot.pdf`.



The rest of the arguments function uses is self-descriptive and is based on the ones from the `Plots.jl` package. Default values of arguments are adjusted to the needs of the *publication*.

source

References

[1] Mikita Hradovich, Adam Kasperski, and Paweł Zieliński. Robust recoverable 0-1 optimization problems under polyhedral uncertainty. *European Journal of Operational Research*, 278(1):136–148, 2019.