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 w hich computing results for adversarial low er bound, recoverable low er bound, selection low er bound or lagrangian low er bound makes sense.

Arguments

- `pd` : an instance of `ProblemDescriptor`

source

`RobRecSolver.hasEqualCardinalityProperty` — *Function*.

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

Returns w hether 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, w hich 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 w ith 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`, w eights `w` and overall w eight 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 w ith one parameter, w hich 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 w ith one parameter, w hich 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)` where

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, which can be allocated to the second stage costs
5. `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

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 with problem size `n` specified by the list `ns` and minimum assignment problem with 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, where 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, where 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 with name `filename` .

Examples:

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

The above command will create file `item_prices.csv` with the following content:

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

source

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

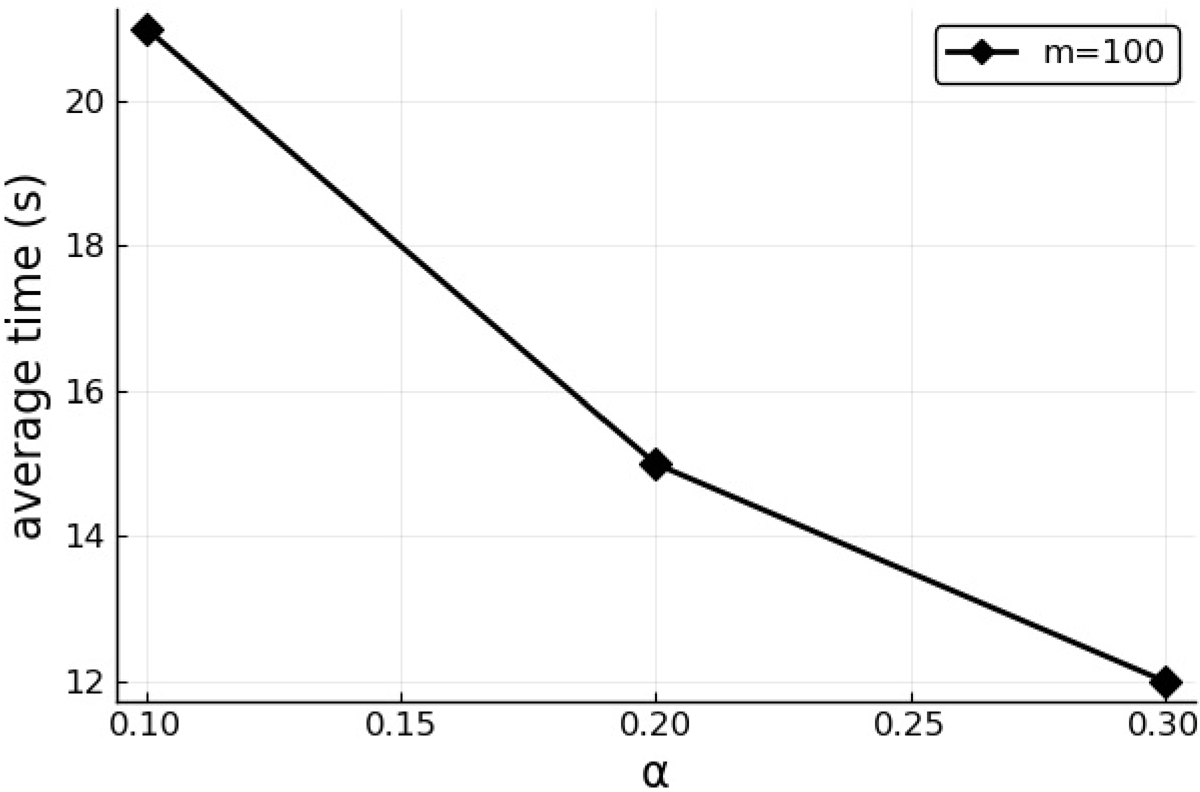
```
drawAndSavePlot(filename, x, ys, xlabel, ylabel, yslabels; linewidth=2, linestyles = [:solid :dash :dashdot :dot :solid], sl
```

Draws plot and saves it to PDF file with 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 will draw the plot shown 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