Package 'prioriactions'

February 9, 2022

Type Package

Version 0.4

Title Multi-Action Conservation Planning

Description This uses a mixed integer mathematical programming (MIP) approach for building and solving multi-action planning problems, where the goal is to find an optimal combination of management actions that abate threats, in an efficient way while accounting for spatial aspects. Thus, optimizing the connectivity and conservation effectiveness of the prioritized units and of the deployed actions. The package is capable of handling different commercial (gurobi) and non-commercial (symphony) MIP solvers. Gurobi optimization solver can be installed using comprehensive instructions in the gurobi installation vignette of the prioritizr package (available in <https://prioritizr.net/articles/gurobi_installation_guide.html>). Methods used in the package refers to Salgado-Rojas et al. (2020) <doi:10.1016/j.ecolmodel.2019.108901>, Beyer et al. (2016) <doi:10.1016/j.ecolmodel.2016.02.005>, Cattarino et al. (2015) <doi:10.1371/journal.pone.0128027> and Watts et al. (2009) <doi:10.1016/j.envsoft.2009.06.005>. See the prioriactions website for more information, documentations and examples.

Depends R (>= 3.5.0)

Imports assertthat (>= 0.2.0), Matrix, proto, magrittr, Rsymphony, tidyr, dplyr, Rcpp, rlang

Suggests knitr, gurobi (>= 9.0), roxygen2, rmarkdown, testthat (>= 3.0.0), raster, tmap, sp, viridis, markdown, data.table, purrr, readr, slam, tibble, methods

LinkingTo Rcpp, RcppArmadillo (>= 0.10.1.0.0), BH

Encoding UTF-8

LazyData true

SystemRequirements C++11

License GPL (>= 2)

Language en-US

RoxygenNote 7.1.2

URL https://prioriactions.github.io/prioriactions/,
https://github.com/prioriactions/prioriactions

BugReports https://github.com/prioriactions/prioriactions/issues

VignetteBuilder knitr

Collate 'RcppExports.R' 'internal.R' 'data-class.R' 'writeOutputs.R'
'optimizationProblem-class.R' 'presolve.R' 'evalBlm.R'
'problem_modifier.R' 'evalBudget.R' 'evalTarget.R'
'getActions.R' 'getConnectivityPenalty.R' 'getCost.R'
'getModelInfo.R' 'getPerformance.R' 'getPotentialBenefit.R'
'getSolutionBenefit.R' 'inputData.R' 'package.R'
'portfolio-class.R' 'print.R' 'problem.R' 'show.R' 'simData.R'
'solution-class.R' 'solve.R' 'utils-pipe.R' 'zzz.R'

Config/testthat/edition 3

NeedsCompilation yes

Author Jose Salgado-Rojas [aut, cre],
Irlanda Ceballos-Fuentealba [aut],
Virgilio Hermoso [aut],
Eduardo Alvarez-Miranda [aut],
Jordi Garcia-Gonzalo [aut]

Maintainer Jose Salgado-Rojas <jose.salgado.rojas@hotmail.com>

Repository CRAN

Date/Publication 2022-02-09 22:30:06 UTC

R topics documented:

`data-class` *Data class*

Description

This class is used to represent data of the instances of the corresponding multi-action planning problem. It includes several methods for retrieving the information of the instance (such as the spatial allocation of threats and species, the cost of management actions or the structure of the spatial connectivity across the area where the planning is carried out. This class is created using the `inputData()` function.

Value

No return value.

Fields

data `list` object containing data.

Methods

getActionsAmount(): `integer`. Number of possible actions.

getData(`character` **name):** `data.frame()`. Object stored in the `data` field with the corresponding `name`. The argument `name` indicates the name of arguments of the `problem` function ("pu", "features", "dist_features", "threats", "dist_threats", "sensitivity" or "boundary").

getFeatureAmount(): `integer`. Number of features.

getFeatureNames(): `character`. Names of features.

getMonitoringCosts(): `numeric` `vector()`. Cost of monitoring each planning unit.

getPlanningUnitsAmount(): `integer`. Number of planning units.

getActionCosts(): `numeric` `vector()`. Cost of actions each planning unit and threat.

getThreatNames(): `character`. Names of threats.

getThreatsAmount(): `integer`. Number of threats.

print(): Print basic information of the data instance.

show(): Call print method.

Examples

```
## set seed for reproducibility
set.seed(14)

## Set prioriactions path
prioriactions_path <- system.file("extdata/example_input/", package = "prioriactions")

## Load in planning unit data
pu_data <- data.table::fread(paste0(prioriactions_path,"/pu.dat"),
                             data.table = FALSE)
head(pu_data)

## Load in feature data
features_data <- data.table::fread(paste0(prioriactions_path,"/features.dat"),
                                   data.table = FALSE)
head(features_data)

## Load in planning unit vs feature data
dist_features_data <- data.table::fread(paste0(prioriactions_path,"/dist_features.dat"),
                                        data.table = FALSE)
head(dist_features_data)

## Load in the threats data
threats_data <- data.table::fread(paste0(prioriactions_path,"/threats.dat"),
                                  data.table = FALSE)
head(threats_data)

## Load in the threats distribution data
dist_threats_data <- data.table::fread(paste0(prioriactions_path,"/dist_threats.dat"),
                                       data.table = FALSE)
head(dist_threats_data)

## Load in the sensitivity data
sensitivity_data <- data.table::fread(paste0(prioriactions_path,"/sensitivity.dat"),
                                      data.table = FALSE)
head(sensitivity_data)

## Load in the boundary data
boundary_data <- data.table::fread(paste0(prioriactions_path,"/boundary.dat"),
                                   data.table = FALSE)
head(boundary_data)

## Create instance
problem_data <- inputData(
  pu = pu_data, features = features_data, dist_features = dist_features_data,
 dist_threats = dist_threats_data, threats = threats_data, sensitivity = sensitivity_data,
  boundary = boundary_data
)

## Summary
print(problem_data)
```

```
## Use class methods
problem_data$getData("features")

problem_data$getFeatureAmount()

problem_data$getFeatureNames()

problem_data$getMonitoringCosts()

problem_data$getPlanningUnitsAmount()

problem_data$getActionCosts()

problem_data$getThreatNames()

problem_data$getThreatsAmount()

problem_data$print()
```

`evalBlm` *Evaluate multiple blm values*

Description

Return one solution per instance for different values of blm. Like `prioriactions()` function, it inherits all arguments from `inputData()`, `problem()` and `solve()`.

Usage

```
evalBlm(values = c(), ...)
```

Arguments

`values`	`numeric`. Values of blm to verify. More than one value is needed.
`...`	arguments inherited from `inputData()`, `problem()` and `solve()` functions.

Details

`evalblm()` creates and solves multiple instances, of the corresponding multi-actions planning problem, for different values of blm. Alternatively, this could be obtained by executing function `prioriactions()` or by steps the `inputData()`, `problem()` and `solve()` functions; using, in each run, different blm values. However, the `evalblm()` function has two advantages with respect to this manual approach: : 1) it is more efficient to create the models (this is because the model is created just once and, at each iteration, only the blm values are updated); and 2) the output is a portfolio object, which allows obtaining information about the group of solutions (including all *get* functions).

Value

An object of class portfolio.

Examples

```
# set seed for reproducibility
set.seed(14)

## Create model and solve
port <- evalBlm(pu = sim_pu_data, features = sim_features_data,
                dist_features = sim_dist_features_data,
                threats = sim_threats_data,
                dist_threats = sim_dist_threats_data,
                sensitivity = sim_sensitivity_data,
                boundary = sim_boundary_data,
                values = c(0.0, 0.01, 0.02, 0.03),
                model_type = "minimizeCosts",
                time_limit = 50,
                output_file = FALSE,
                cores = 2)

getConnectivityPenalty(port)
```

`evalBudget` *Evaluate multiple budget values*

Description

Return one solution per instance for different values of budgets. This function assumes that the *maximizeBenefits* option is being used (note that the *minimizeCosts* option does not require setting a maximum budget). Like `prioriactions()` function, it inherits all arguments from `inputData()`, `problem()` and `solve()`.

Usage

```
evalBudget(values = c(), ...)
```

Arguments

`values`	`numeric`. Values of budget to verify. More than one value is needed.
`...`	arguments inherited from `inputData()`, `problem()`, and `solve()` functions.

Details

`evalBudget()` creates and solves multiple instances, of the corresponding multi-actions planning problem, for different values of maximum budgets. Alternatively, this could be obtained by executing function `prioriactions()` or by steps the `inputData()`, `problem()` and `solve()` functions; using, in each run, different budgets values. However, the `evalBudget()` function has two advantages with respect to this manual approach: : 1) it is more efficient to create the models (this is

because the model is created just once and, at each iteration, only the budget values are updated); and 2) the output is a portfolio object, which allows obtaining information about the group of solutions (including all *get* functions).

Value

An object of class portfolio.

Examples

```
# set seed for reproducibility
set.seed(14)

## Create model and solve
port <- evalBudget(pu = sim_pu_data, features = sim_features_data,
                dist_features = sim_dist_features_data,
                threats = sim_threats_data,
                dist_threats = sim_dist_threats_data,
                sensitivity = sim_sensitivity_data,
                boundary = sim_boundary_data,
                values = c(1, 10, 50, 100),
                time_limit = 50,
                output_file = FALSE,
                cores = 2)

getSolutionBenefit(port)
```

---

evalTarget *Evaluate multiple target values*

Description

Return one solution per instance for different targets values. This function assumes that the *minimizeCosts* model is being used. As well as the `prioriactions()` function, it inherits all arguments from `inputData()`, `problem()` and `solve()`.

Usage

```
evalTarget(values = c(), ...)
```

Arguments

`values`	`numeric`. Proportion of maximum value of benefits to verify (both recovery and conservation benefits). This information can be obtained with `getPotentialBenefit()` function. More than one value is needed.
`...`	arguments inherited from `inputData()`, `problem()`, and `solve()` functions.

Details

`evalTarget()` creates and solves multiple instances, of the corresponding multi-actions planning problem, for different proportions of maximum benefit values as target values. It is assumed that the same proportion is applied for the maximum benefit in recovery and conservation. Alternatively, this could be obtained by executing function `prioriactions()` or by steps the `inputData()`, `problem()` and `solve()` functions; using, in each run, different targets values. However, the `evalTarget()` function has two advantages with respect to this manual approach: : 1) it is more efficient to create the models (this is because the model is created just once and, at each iteration, only the target values are updated); and 2) the output is a portfolio object, which allows obtaining information about the group of solutions (including all *get* functions).

Value

An object of class portfolio.

Examples

```
# set seed for reproducibility
set.seed(14)

## Create model and solve
port <- evalTarget(pu = sim_pu_data, features = sim_features_data,
                dist_features = sim_dist_features_data,
                threats = sim_threats_data,
                dist_threats = sim_dist_threats_data,
                sensitivity = sim_sensitivity_data,
                boundary = sim_boundary_data,
                values = c(0.1, 0.3, 0.5),
                time_limit = 50,
                output_file = FALSE,
                cores = 2)

getCost(port)
```

---

getActions *Extract action information*

Description

Returns the spatial deployment of the actions for each planning unit of the corresponding solution.

Usage

```
getActions(x, format = "wide")
```

Arguments

x	solution or portfolio object.
format	character. Output format of the action matrix; wide format shows one column per action, while large format shows four columns: solution_name, pu, action and solution.

Details

getActions() function assumes that actions can be of three types:

1. **to abate specific threats**: these actions have the *id* corresponding to the threat to be abate.
2. **to conservation**: that indicates if the planning unit is selected to conservative any feature that is not threatened.
3. **to connectivity**: that indicates if the planning unit is selected only by connectivity (i.e. without performing conservation actions or actions against a threat in said unit).

Value

data.frame.

Examples

```
# set seed for reproducibility
set.seed(14)

## Load data
data(sim_pu_data, sim_features_data, sim_dist_features_data,
sim_threats_data, sim_dist_threats_data, sim_sensitivity_data,
sim_boundary_data)

## Create instance
problem_data <- inputData(
  pu = sim_pu_data, features = sim_features_data, dist_features = sim_dist_features_data,
  threats = sim_threats_data, dist_threats = sim_dist_threats_data,
  sensitivity = sim_sensitivity_data, boundary = sim_boundary_data
)

## Create optimization model
problem_model <- problem(x = problem_data)

## Solve the optimization model
s <- solve(a = problem_model, time_limit = 2, output_file = FALSE, cores = 2)

# get actions information in large format
actions <- getActions(s, format = "large")
head(actions)

# get actions information in wide format
actions <- getActions(s, format = "wide")
```

```
head(actions)
```

`getConnectivityPenalty` *Extract connectivity penalty values*

Description

Provides the connectivity penalty value for all actions and planning units in a solution.

Usage

```
getConnectivityPenalty(x)
```

Arguments

x	solution or portfolio object.

Details

The connectivity penalty among is calculated as the sum of all connectivity penalties by each action and planning unit in the solution. This can be expressed mathematically for a set of planning units I indexed by i and j, and a set of threats K indexed by k as:

$$\sum_{k \in K} \sum_{i \in I_k} \sum_{j \in I_k} x_{ik}(1 - x_{jk}) cv_{ij}$$

Where, x_{ik} is the decisions variable that specify whether an action has been selected to abate threat k in planning unit i (1) or not (0), cv_{ij} is the connectivity penalty that applies when a solution contains planning unit i but not j o viceversa.

Note that there is an action per threat, so it is assumed that the index of the threat coincides with the index of the action used to abate it.

Value

data.frame.

Examples

```
# set seed for reproducibility
set.seed(14)

## Load data
data(sim_pu_data, sim_features_data, sim_dist_features_data,
sim_threats_data, sim_dist_threats_data, sim_sensitivity_data,
```

```
sim_boundary_data)

## Create data instance
problem_data <- inputData(
  pu = sim_pu_data, features = sim_features_data, dist_features = sim_dist_features_data,
  threats = sim_threats_data, dist_threats = sim_dist_threats_data,
  sensitivity = sim_sensitivity_data, boundary = sim_boundary_data
)

## Create optimization model
problem_model <- problem(x = problem_data, blm = 0.03)

## Solve the optimization model
s <- solve(a = problem_model, time_limit = 2, output_file = FALSE, cores = 2)

# get connectivity penalty values
getConnectivityPenalty(s)
```

`getCost` *Extract cost values*

Description

Provides the sum of costs to actions and monitoring applied in a solution.

Usage

```
getCost(x)
```

Arguments

x	solution or portfolio object.

Details

The cost value is calculated as the sum of all the individual costs of actions and monitoring carried out in each of the planning units. This can be expressed mathematically for a set of planning units I indexed by i, and a set of threats K indexed by k as:

$$actions = \sum_{i \in I} \sum_{k \in K_i} x_{ik} c_{ik}$$

$$monitoring = \sum_{i \in I} x_{i.} c_i^{'}$$

Where, x_{ik} is the decisions variable that specify whether an action has been selected to abate threat k in planning unit i (1) or not (0), c_{ik} is the action cost to abate threat k in planning unit i and $c_i^{'}$ is

the monitoring cost of planning unit i. The cost of monitoring is applied to all planning units where some type of action has been selected (conservation action, to abate threats or connectivity).

Note that there is an action per threat, so it is assumed that the index of the threat coincides with the index of the action used to abate it.

Value

data.frame.

Examples

```
# set seed for reproducibility
set.seed(14)

## Load data
data(sim_pu_data, sim_features_data, sim_dist_features_data,
sim_threats_data, sim_dist_threats_data, sim_sensitivity_data,
sim_boundary_data)

## Create data instance
problem_data <- inputData(
  pu = sim_pu_data, features = sim_features_data, dist_features = sim_dist_features_data,
  threats = sim_threats_data, dist_threats = sim_dist_threats_data,
  sensitivity = sim_sensitivity_data, boundary = sim_boundary_data
)

## Create optimization model
problem_model <- problem(x = problem_data)

## Solve the optimization model
s <- solve(a = problem_model, time_limit = 2, output_file = FALSE, cores = 2)

## Get costs
getCost(s)
```

---

getModelInfo *Extract general information about mathematical model*

Description

Provides general information about the mathematical model.

Usage

```
getModelInfo(x)
```

Arguments

x	optimizationProblem, solution or portfolio object.

Details

`getModelInfo()` function returns five specific fields:

1. **solution_name**: indicates the name of the solution, by default is *sol*.
2. **model_sense**: returns the optimization sense (i.e., it indicates whether the objective function is minimized or maximize).
3. **n_constraints**: returns the number of constraints in the corresponding mathematical optimization model.
4. **n_variables**: returns the number of variables in the corresponding mathematical optimization model.
5. **size**: returns the size of the constraints' coefficients matrix A number of constraints and number of variables).

Value

data.frame.

Examples

```
# set seed for reproducibility
set.seed(14)

## Load data
data(sim_pu_data, sim_features_data, sim_dist_features_data,
sim_threats_data, sim_dist_threats_data, sim_sensitivity_data,
sim_boundary_data)

## Create data instance
problem_data <- inputData(
  pu = sim_pu_data, features = sim_features_data, dist_features = sim_dist_features_data,
  threats = sim_threats_data, dist_threats = sim_dist_threats_data,
  sensitivity = sim_sensitivity_data, boundary = sim_boundary_data
)

## Create optimization model
problem_model <- problem(x = problem_data, blm = 1)

# get model information
getModelInfo(problem_model)
```

`getPerformance` *Extract general information about solution*

Description

Provides general information about the process of solving.

Usage

```
getPerformance(x)
```

Arguments

x	solution or portfolio object.

Details

`getPerformance()` function returns five specific fields:

1. **solution_name**: indicates the name of the solution, by default is *sol*.
2. **objective_value**: indicates the value of the objective function of a given solution. This value depends on the type of model solved (more information in the `problem()` function).
3. **gap**: returns the relative MIP optimality gap of a solution. It is measured as the ratio between the objective function induced by the best known (primal solution) integer solution and the objective function induced by the best node in the search tree (dual solution).
4. **solving_time**: indicates the solving time of mathematical model.
5. **status**: provides the status of solver at the end of the optimization period. This can have six states:

- *Optimal solution (according to gap tolerance)* : When the resolution of the model stop when the quality of the solution (gap) is less than or equal to gap_limit (parameter of the `solve()` function).
- *No solution (model was proven to be infeasible or unbounded)*: When the model is infeasible.
- *Feasible solution (according to time limit)*: When the resolution of the model stops when a time_limit has been reached finding a feasible solution (parameter of the `solve()` function).
- *No solution (according to time limit)*: When the resolution of the model stops when a time_limit has been reached without finding a feasible solution (parameter of the `solve()` function).
- *First feasible solution*: When the resolution of the model stops when it has found the first feasible solution (solution_limit = TRUE parameter in `solve()` function).
- *No solution information is available*: For any other case.

Value

data.frame.

Examples

```
# set seed for reproducibility
set.seed(14)

## Load data
data(sim_pu_data, sim_features_data, sim_dist_features_data,
sim_threats_data, sim_dist_threats_data, sim_sensitivity_data,
sim_boundary_data)

## Create data instance
problem_data <- inputData(
  pu = sim_pu_data, features = sim_features_data, dist_features = sim_dist_features_data,
  threats = sim_threats_data, dist_threats = sim_dist_threats_data,
  sensitivity = sim_sensitivity_data, boundary = sim_boundary_data
)

## Create optimization model
problem_model <- problem(x = problem_data, blm = 1)

## Solve the optimization model
s <- solve(a = problem_model, time_limit = 2, output_file = FALSE, cores = 2)

# get solution gap
getPerformance(s)
```

---

getPotentialBenefit *Extract potential benefit of features*

Description

Provides the maximum values of benefits to achieve for each feature given a set of data inputs.

Usage

```
getPotentialBenefit(x)
```

Arguments

x	data-class object.

Details

For a given feature s, let I_s be the set of planning units associated with s, let r_{is} is the amount of feature s in planning unit i, let K_s be the set of threats associated with s, and let K_i be the set of threats associated with i. The local benefit associated with s in a unit i is given by:

$$b_{is} = p_{is} r_{is} b_{is} = \frac{\sum_{k \in K_i \cap K_s} x_{ik}}{|K_i \cap K_s|} r_{is}$$

Where x_{ik} is a decision variable such that $x_{ik} = 1$ if an action againts threat k is applied in unit i, and $x_{ik} = 0$, otherwise. This expression for the probability of persistence of the feature (p_{is}) is defined only for the cases where we work with values of binary intensities (presence or absence of threats). See the sensitivities vignette to know the work with continuous intensities.

While the total benefit is calculated as the sum of the local benefits per feature:

$$b_s = \sum_{i \in I_s} \frac{\sum_{k \in K_i \cap K_s} x_{ik}}{|K_i \cap K_s|} r_{is}$$

Since the potential benefit is being calculated, all variables x_{ik} are assumed to be equal to 1; that is, all possible actions are carried out, and only those that have a **lock-out** status are kept out of the planning (see `inputData()` function for more information).

Value

data.frame.

Examples

```
# set seed for reproducibility
set.seed(14)

## Load data
data(sim_pu_data, sim_features_data, sim_dist_features_data,
sim_threats_data, sim_dist_threats_data, sim_sensitivity_data,
sim_boundary_data)

## Create data instance
problem_data <- inputData(
  pu = sim_pu_data, features = sim_features_data, dist_features = sim_dist_features_data,
  threats = sim_threats_data, dist_threats = sim_dist_threats_data,
  sensitivity = sim_sensitivity_data, boundary = sim_boundary_data
)

## Get maximum benefits to obtain
getPotentialBenefit(problem_data)
```

`getSolutionBenefit` *Extract benefit values*

Description

Returns the total benefit induced by the corresponding solution. The total benefit is computed as the sum of the benefits obtained, for all features, across all the units in the planning area.

Usage

```
getSolutionBenefit(x, type = "total")
```

Arguments

`x`	`Solution-class` or `Portfolio-class`.
`type`	`character`. Output format of the benefits matrix; `total` shows the total benefit by feature, while `local` format shows the benefit achieved per feature and planning unit.

Details

For a given feature s, let I_s be the set of planning units associated with s, let r_{is} is the amount of feature s in planning unit i, let K_s be the set of threats associated with s, and let K_i be the set of threats associated with i. The local benefit associated with s in a unit i is given by:

$$b_{is} = p_{is} r_{is} b_{is} = \frac{\sum_{k \in K_i \cap K_s} x_{ik}}{|K_i \cap K_s|} r_{is}$$

Where x_{ik} is a decision variable such that $x_{ik} = 1$ if an action againts threat k is applied in unit i, and $x_{ik} = 0$, otherwise. This expression for the probability of persistence of the feature (p_{is}) is defined only for the cases where we work with values of binary intensities (presence or absence of threats). See the sensitivities vignette to know the work with continuous intensities.

While the total benefit is calculated as the sum of the local benefits per feature:

$$b_s = \sum_{i \in I_s} \frac{\sum_{k \in K_i \cap K_s} x_{ik}}{|K_i \cap K_s|} r_{is}$$

Value

data.frame.

Examples

```
# set seed for reproducibility
set.seed(14)

## Load data
data(sim_pu_data, sim_features_data, sim_dist_features_data,
sim_threats_data, sim_dist_threats_data, sim_sensitivity_data,
sim_boundary_data)

## Create data instance
problem_data <- inputData(
  pu = sim_pu_data, features = sim_features_data, dist_features = sim_dist_features_data,
  threats = sim_threats_data, dist_threats = sim_dist_threats_data,
  sensitivity = sim_sensitivity_data, boundary = sim_boundary_data
)

## Get maximum benefits to obtain
getPotentialBenefit(problem_data)

## Create optimization model
problem_model <- problem(x = problem_data)

## Solve the optimization model
s <- solve(a = problem_model, time_limit = 2, output_file = FALSE, cores = 2)

# get local benefits of solution
local_benefit <- getSolutionBenefit(s, type = "local")
head(local_benefit)

# get total benefits of solution
total_benefit <- getSolutionBenefit(s, type = "total")
head(total_benefit)
```

`inputData` *Creates the multi-action planning problem*

Description

Create the data object with information about the multi-action conservation planning problem. This function is used to specify all the data that defines the spatial prioritization problem (planning units data, feature data, threats data, and their spatial distributions.)

Usage

```
inputData(pu, features, dist_features, threats, dist_threats, ...)
```

```
## S4 method for signature
## 'data.frame,data.frame,data.frame,data.frame,data.frame'
inputData(
  pu,
  features,
  dist_features,
  threats,
  dist_threats,
  sensitivity = NULL,
  boundary = NULL
)
```

Arguments

pu	Object of class `data.frame()` that specifies the planning units (PU) of the corresponding instance and their corresponding monitoring cost and status. Each row corresponds to a different planning unit. This file is inherited from the *pu.dat* in *Marxan*. It must contain the following columns: `id integer` unique identifier for each planning unit. `monitoring_cost numeric` cost of including each planning unit in the reserve system. `status integer` (**optional**) value that indicate if each planning unit should be available to be selected (0), *locked-in* (2) as part of the solution, or *locked-out* (3) and excluded from the solution.
features	Object of class `data.frame()` that specifies the conservation features to consider in the optimization problem. Each row corresponds to a different feature. This file is inherited from marxan's *spec.dat*. The `prioriactions` package supports two types of purposes when optimizing: focus on recovery of features threatened (through the **recovery target**), where only take into account benefits when taking action against threats and there is no benefit when selecting planning units where the features are not threatened; or include the benefits of the features sites where they are not threatened (through the **conservation target**). Note that by default only information on recovery targets is necessary, while conservation targets equal to zero are assumed. The maximum values of benefits to achieve both recovery and conservation per feature can be verified with the `getPotentialBenefit()` function. For more information on the implications of these targets in the solutions see the recovery vignette. This file must contain the following columns: `id integer` unique identifier for each conservation feature. `target_recovery numeric` amount of recovery target to achieve for each conservation feature. This field is **required** if a `minimizeCosts` model is used. `target_conservation numeric` (**optional**) amount of conservation target to achieve for each conservation feature. This field is used only if a model of the type `minimizeCosts` is applied. `name character` (**optional**) name for each conservation feature.

`dist_features` Object of class `data.frame()` that specifies the spatial distribution of conservation features across planning units. Each row corresponds to a combination of planning unit and feature. This file is inherited from marxan's *puvspr.dat*. It must contain the following columns:

`pu integer` *id* of a planning unit where the conservation feature listed on the same row occurs.

`feature integer` *id* of each conservation feature.

`amount numeric` amount of the feature in the planning unit. Set to 1 to work with presence/absence.

`threats` Object of class `data.frame()` that specifies the threats to consider in the optimization exercise. Each row corresponds to a different threats. It must contain the following columns:

`id integer` unique identifier for each threat.

`blm_actions numeric` (**optional**) penalty of connectivity between threats. Default is 0.

`name character` (**optional**) name for each threat.

`dist_threats` Object of class `data.frame()` that specifies the spatial distribution of threats across planning units. Each row corresponds to a combination of planning unit and threat. It must contain the following columns:

`pu integer` *id* of a planning unit where the threat listed on the same row occurs.

`threat integer` *id* of each threat.

`amount numeric` amount of the threat in the planning unit. Set to 1 to work with presence/absence. Continuous amount values require that feature sensitivities to threats be established (more info in sensitivities vignette).

`action_cost numeric` cost of an action to abate the threat in each planning unit.

`status integer` (**optional**) value that indicates if each action to abate the threat is available to be selected (0), *locked-in* (2) as part of the solution, or *locked-out* (3) and therefore excluded from the solution.

`...` Unused arguments, reserved for future expansion.

`sensitivity` (**optional**) Object of class `data.frame()` that specifies the sensitivity of each feature to each threat. Each row corresponds to a combination of feature and threat. If not informed, all features are assumed to be sensitive to all threats.

Sensitivity can be parameterized in two ways: **binary**; the feature is sensitive or not, or **continuous**; with response curves of the probability of persistence of the features to threats. For the first case, it is only necessary to indicate the ids of the threats and the respective features sensitive to them. In the second case, the response can be parameterized through four values: *a*, *b*, *c* and *d*. See sensitivities vignette for more information on continuous sensitivities. Then, the sensitivity input must contain the following columns:

`feature integer` *id* of each conservation feature.

`threat integer` *id* of each threat.

`a numeric` (**optional**) the minimum intensity of the threat at which the features probability of persistence starts to decline. The more sensitive the feature is to the threat, the lowest this value will be. Default is 0.

`b` `numeric` (**optional**) the value of intensity of the threat over which the feature has a probability of persistence of 0. If it is not established,it is assumed as the **maximum value of the threat across all planning units** in the study area. Note that this might overestimate the sensitivity of features to threats, as they will only be assumed to disappear from planning units if the threats reach the maximum intensity value in the study area.

`c` `numeric` (**optional**) minimum probability of persistence of a features when a threat reaches its maximum intensity value. Default is 0.

`d` `numeric` (**optional**) maximum probability of persistence of a features in absence of a given threat. Default is 1.

Note that optional parameters *a*, *b*, *c* and *d* can be provided independently.

`boundary` (**optional**) Object of class `data.frame()` that specifies the spatial relationship between pair of planning units. Each row corresponds to a combination of planning unit. This file is inherited from marxan's *bound.dat*. It must contain the following columns:

`id1` `integer` *id* of each planning unit.

`id2` `integer` *id* of each planning unit.

`boundary` `numeric` penalty applied in the objective function when only one of the planning units is present in the solution.

Value

An object of class data.

References

- Ball I, Possingham H, Watts, M. *Marxan and relatives: software for spatial conservation prioritization*. Spatial conservation prioritisation: quantitative methods and computational tools 2009.

See Also

For more information on the correct format for *Marxan* input data, see the official *Marxan* website and Ball *et al.* (2009).

Examples

```
## set seed for reproducibility
set.seed(14)

## Set prioriactions path
prioriactions_path <- system.file("extdata/example_input/", package = "prioriactions")

## Load in planning unit data
pu_data <- data.table::fread(paste0(prioriactions_path,"/pu.dat"),
                             data.table = FALSE)
head(pu_data)

## Load in feature data
```

```
features_data <- data.table::fread(paste0(prioriactions_path,"/features.dat"),
                                   data.table = FALSE)
head(features_data)

## Load in planning unit vs feature data
dist_features_data <- data.table::fread(paste0(prioriactions_path,"/dist_features.dat"),
                                        data.table = FALSE)
head(dist_features_data)

## Load in the threats data
threats_data <- data.table::fread(paste0(prioriactions_path,"/threats.dat"),
                                  data.table = FALSE)
head(threats_data)

## Load in the threats distribution data
dist_threats_data <- data.table::fread(paste0(prioriactions_path,"/dist_threats.dat"),
                                       data.table = FALSE)
head(dist_threats_data)

## Load in the sensitivity data
sensitivity_data <- data.table::fread(paste0(prioriactions_path,"/sensitivity.dat"),
                                      data.table = FALSE)
head(sensitivity_data)

## Load in the boundary data
boundary_data <- data.table::fread(paste0(prioriactions_path,"/boundary.dat"),
                                   data.table = FALSE)
head(boundary_data)

## Create data instance
problem_data <- inputData(
  pu = sim_pu_data, features = sim_features_data, dist_features = sim_dist_features_data,
  threats = sim_threats_data, dist_threats = sim_dist_threats_data,
  sensitivity = sim_sensitivity_data, boundary = sim_boundary_data
)

## Summary
print(problem_data)
```

`optimizationProblem-class`

Optimization problem class

Description

This class encodes the corresponding optimization model. It is created using `problem()` function.

Value

No return value.

Fields

$data `list` object containing data of the mathematical model.

$ConservationClass object of class `data-class()` that contains the data input.

Methods

getData(`character` **name)** `vector()`. Object stored in the `data` field with the corresponding `name`. The data correspond to the different parts of the mathematical model. The argument `name` can be made to the following: "obj", "rhs", "sense", "vtype", "A", "bounds" or "modelsense".

getDataList() `list()` of `vector()`. Object stored in the `data`. It contains all information relative to the mathematical model, such as "obj", "rhs", etc.

print() Print basic information of the optimization model.

show() Call print method.

Examples

```
# set seed for reproducibility
set.seed(14)

## Load data
data(sim_pu_data, sim_features_data, sim_dist_features_data,
sim_threats_data, sim_dist_threats_data, sim_sensitivity_data,
sim_boundary_data)

## Create data instance
problem_data <- inputData(
  pu = sim_pu_data, features = sim_features_data, dist_features = sim_dist_features_data,
  threats = sim_threats_data, dist_threats = sim_dist_threats_data,
  sensitivity = sim_sensitivity_data, boundary = sim_boundary_data
)

## Create optimization model
problem_model <- problem(x = problem_data, blm = 1)

## Use class methods
head(problem_model$getData("obj"))

problem_model$print()
```

`portfolio-class` *Portfolio class*

Description

This class encodes for the solutions obtained when solving multiple instances. This includes several methods to obtain information about both the optimization process and the solution associated with the planning units and conservation actions. It is created using the *eval* functions (e.g. `evalTarget()` or `evalBudget()`).

Value

No return value.

Fields

$data `list`. Object containing data on the results of the optimization process.

Methods

getNames() `character`. Label indicating the name of solutions.

print() Print basic information of the model solution.

show() Call print method.

Examples

```
# set seed for reproducibility
set.seed(14)

## Create model and solve
port <- evalBlm(pu = sim_pu_data, features = sim_features_data,
                dist_features = sim_dist_features_data,
                threats = sim_threats_data,
                dist_threats = sim_dist_threats_data,
                sensitivity = sim_sensitivity_data,
                boundary = sim_boundary_data,
                values = c(0.0, 0.01, 0.02, 0.03),
                model_type = "minimizeCosts",
                time_limit = 50,
                output_file = FALSE, cores = 2)

## Use class methods
port$getNames()

port$print()
```

`print` *Print*

Description

Displays information about an object.

Usage

```
## S3 method for class 'Data'
print(x, ...)

## S3 method for class 'OptimizationProblem'
print(x, ...)

## S3 method for class 'Solution'
print(x, ...)

## S3 method for class 'Portfolio'
print(x, ...)
```

Arguments

x	Any object.
...	Not used.

Value

None.

See Also

base::print().

---

prioriactions *Create and solve multi-actions planning problems*

Description

Create and solve a multi-actions planning problem. It can be used instead of following the sequence of the inputData(), problem() and solve() functions.

Usage

```
prioriactions(...)
```

Arguments

...	arguments inherited from inputData(), problem() and solve() functions.

Value

An object of class solution.

Examples

```
## This example uses input files included into package.

## set seed for reproducibility
set.seed(14)

## Load data
data(sim_pu_data, sim_features_data, sim_dist_features_data,
sim_threats_data, sim_dist_threats_data, sim_sensitivity_data,
sim_boundary_data)

## Create data instance
s <- prioriactions(pu = sim_pu_data, features = sim_features_data,
                dist_features = sim_dist_features_data,
                threats = sim_threats_data,
                dist_threats = sim_dist_threats_data,
                sensitivity = sim_sensitivity_data,
                boundary = sim_boundary_data,
                model_type = "minimizeCosts",
                time_limit = 50,
                output_file = FALSE,
                cores = 2)

print(s)
```

`problem` *Create mathematical model*

Description

Create an optimization model for the multi-action conservation planning problem, following the mathematical formulations used in Salgado-Rojas *et al.* (2020).

Usage

```
problem(
  x,
  model_type = "minimizeCosts",
  budget = 0,
  blm = 0,
  curve = 1,
  segments = 3
)
```

Arguments

x	data object. Data used in a problem of prioritization of multiple conservation actions. This object must be created using the `problem()` function.
model_type	`character`. Name of the type of model to create. With two possible values: `minimizeCosts` and `maximizeBenefits`.
budget	`numeric`. Maximum budget allowed. This field is used only if a model of the type `maximizeBenefits` is applied.
blm	`numeric`. Weight factor applied to the sum of connectivity penalties for missed connections in a solution, similar to Boundary Length Modifier (BLM) in *Marxan*. This argument only has an effect when the `boundary` is available.
curve	`integer`. Type of continuous curve used to represent benefit expression. It can be a linear (`1`), quadratic (`2`) or cubic (`3`) function. See **Details** for more information.
segments	`integer`. Number of segments (`1`, `2`, `3`, `4` or `5`) used to approximate the non-linear expression (`curve`) in the calculate benefits. See **Details** for more information.

Details

Currently the problem function allows you to create two types of mathematical programming models:

minimize cost (minimizeCosts): This model seeks to find the set of actions that minimizes the overall planning costs, while meeting a set of representation targets for the conservation features.

This model can be expressed mathematically for a set of planning units I indexed by i a set of features S indexed by s, and a set of threats K indexed by k as:

$$\min \sum_{i \in I} \sum_{k \in K_i} x_{ik} c_{ik} + \sum_{i \in I} x_i . c'_i + blm \cdot connectivity s.t. \sum_{i \in I_s} p_{is} r_{is} \geq t_s \forall s \in S$$

Where, x_{ik} is a decisions variable that specifies whether an action to abate threat k in planning unit i has been selected (1) or not (0), c_{ik} is the cost of the action to abate the threat k in the planning unit i, c'_i is the monitoring cost of planning unit i, p_{is} is the probability of persistence of the feature s in the planning unit i (ranging between 0 and 1), r_{is} is the amount of feature s in planning unit i. t_s is the **recovery target** for feature s. In the case of working with **conservation target**, the following constraint is necessary:

$$\sum_{i \in I_s : |K_s \cap K_i| \neq 0} z_{is} r_{is} \geq t'_s \forall s \in S$$

With, z_{is} as the probability of persistence by conservation of the feature s in the planning unit i (ranging between 0 and 1). It is only present when there is no spatial co-occurrence between a feature and its threats (i.e. $|K_s \cap K_i| \neq 0$). In the case of binary threat intensities it is assumed as 1. t'_s is the **conservation target** for feature s.

maximize benefits (maximizeBenefits): The maximize benefits model seeks to find the set of actions that maximizes the sum of benefits of all features, while the cost of performing actions

and monitoring does not exceed a certain budget. Using the terminology presented above, this model can be expressed mathematically as:

$$\max \sum_{i \in I} \sum_{s \in S_i} b_{is} - blm \cdot connectivity s.t. \sum_{i \in I} \sum_{k \in K_i} x_{ik} c_{ik} + \sum_{i \in I} x_i . c_i' \leq budget$$

Where b_{is} is the benefit of the feature s in a planning unit i and it is calculated by multiplying the probability of persistence of the feature in the unit by its corresponding amount, i.e., $b_{is} = p_{is} r_{is}$. When we talk about recovering, the probability of persistence is a measure of the number of actions taken against the threats that affect said feature. For more information on its calculation, see the `getSolutionBenefit()` or `getPotentialBenefit()` functions references.

As a way of including the risk associated with calculating our probability of persistence of the features and in turn, avoiding that many low probabilities of persistence end up reaching the proposed targets, is that we add the `curve` parameter. That incorporates an exponent (values of 1: linear, 2: quadratic or 3: cubic) to the calculation of the probability of persistence. Thus penalizing the low probabilities in the sum of the benefits achieved. Since `prioriactions` works with linear models, we use a piecewise linearization strategy to work with non-linear curves in b_{is}. The `segments` parameter indicates how well the expression approximates the curved used in b_{is}. A higher number implies a better approximation but increases the resolution complexity. Note that for a linear curve (`curve = 1`) it is not necessary to set a `segment` parameter.

Parameters `blm` and `blm_actions` allow controlling the spatial connectivity of the selected units and of the deployed actions, respectively (similar to BLM in Marxan).

Value

An object of class optimizationProblem.

See Also

For more information regarding the arguments `curve` and `segments`, see the supplementary material of Salgado-Rojas *et al.* (2020)..

Examples

```
## This example uses input files included into package.

## set seed for reproducibility
set.seed(14)

## Load data
data(sim_pu_data, sim_features_data, sim_dist_features_data,
sim_threats_data, sim_dist_threats_data, sim_sensitivity_data,
sim_boundary_data)

## Create data instance
problem_data <- inputData(
  pu = sim_pu_data, features = sim_features_data, dist_features = sim_dist_features_data,
  threats = sim_threats_data, dist_threats = sim_dist_threats_data,
  sensitivity = sim_sensitivity_data, boundary = sim_boundary_data
)
```

```
## Create minimizeCosts model
model_min <- problem(x = problem_data, blm = 1, model_type = "minimizeCosts")

#' ## Create maximazeBenefits model
model_max <- problem(x = problem_data, model_type = "maximizeBenefits", budget = 100)
```

---

show *Show*

Description

Displays information about an object.

Usage

```
## S3 method for class 'Data'
show(x, ...)

## S3 method for class 'OptimizationProblem'
show(x, ...)

## S3 method for class 'Solution'
show(x, ...)

## S3 method for class 'Portfolio'
show(x, ...)
```

Arguments

x	Any object.
...	Not used.

Value

None.

See Also

methods::show().

Examples

```
a <- 1:4
show(a)
```

simData *Simulated multi-action planning data*

Description

Simulated data for making prioritizations.

`sim_pu_data` Planning units are represented as tabular data.

`sim_features_data` Features are represented as tabular data.

`sim_dist_features_data` The simulated distribution of four features.

`sim_threats_data` Threats are represented as tabular data.

`sim_dist_threats_data` The simulated threats of two threats.

`sim_sensitivity_data` Sensitivity of features to threats as tabular data.

`sim_boundary_data` Boundary data between one hundred planning units.

Usage

```
data(sim_pu_data)

data(sim_features_data)

data(sim_dist_features_data)

data(sim_threats_data)

data(sim_dist_threats_data)

data(sim_sensitivity_data)

data(sim_boundary_data)
```

Format

sim_pu_data data.frame object.

sim_features_data data.frame object.

sim_dist_features_data data.frame object.

sim_threats_data data.frame object.

sim_dist_threats_data data.frame object.

sim_sensitivity_data data.frame object.

sim_boundary_data data.frame object.

Examples

```
## Not run:
# load data
data(sim_pu_data, sim_features_data, sim_dist_features_data,
sim_threats_data, sim_dist_threats_data, sim_sensitivity_data,
sim_boundary_data)

# plot examples
library(raster)
r <- raster::raster(ncol=10, nrow=10, xmn=0, xmx=10, ymn=0, ymx=10)

# plot cost of pu's
values(r) <- sim_pu_data$monitoring_cost
plot(r)

# plot feature distribution of feature 1
features <- tidyr::spread(data = sim_dist_features_data, key = feature, value = amount, fill = 0)
values(r) <- features$'1'
plot(r)

## End(Not run)
```

---

solution-class *Solution class*

Description

This class is used to represent the solution of the MIP (Mixed-Integer Programming) model. This includes several methods to obtain information about both the optimization process and the solution associated with the planning units and actions. It is created using the `solve()` function.

Value

No return value.

Fields

$data `list`. Object containing data on the results of the optimization process.

Methods

print() Print basic information of the model solution.

show() Call print method.

Examples

```
# set seed for reproducibility
set.seed(14)

## Load data
data(sim_pu_data, sim_features_data, sim_dist_features_data,
sim_threats_data, sim_dist_threats_data, sim_sensitivity_data,
sim_boundary_data)

## Create data instance
problem_data <- inputData(
  pu = sim_pu_data, features = sim_features_data, dist_features = sim_dist_features_data,
  threats = sim_threats_data, dist_threats = sim_dist_threats_data,
  sensitivity = sim_sensitivity_data, boundary = sim_boundary_data
)

## Create optimization model
problem_model <- problem(x = problem_data, blm = 1)

## Solve the optimization model
s <- solve(a = problem_model, time_limit = 5, output_file = FALSE, cores = 2)

## Use class methods

s$print()
```

---

solve *Solve mathematical models*

Description

Solves the optimization model associated with the multi-action conservation planning problem. This function is used to solve the mathematical model created by the `problem()` function.

Usage

```
solve(
  a,
  solver = "",
  gap_limit = 0,
  time_limit = .Machine$integer.max,
  solution_limit = FALSE,
  cores = 2,
  verbose = TRUE,
  name_output_file = "output",
  output_file = TRUE
)
```

Arguments

a	optimizationProblem object. Optimization model created for the problem of prioritization of multiple conservation actions. This object must be created using the problem() function.
solver	string. Name of solver to use to solve the model. The following solvers are supported: "gurobi"(requires the **gurobi** package), and "symphony"(requires the **Rsymphony** package). We recommend using gurobi (for more information on how to obtain an academic license here).
gap_limit	numeric. Value between 0 and 1 that represents the gap to optimality, i.e., a relative number that cause the optimizer to terminate when the difference between the upper and lower objective function bounds is less than the gap times the upper bound. For example, a value of 0.01 will result in the optimizer stopping when the difference between the bounds is 1 percent of the upper bound. Default is 0.0.
time_limit	numeric. Time limit to run the optimizer (in seconds). The solver will return the current best solution when this time limit is exceeded. Default is the maximum integer number of your machine.
solution_limit	logical. Indicates if the solution process should be stopped after the first feasible solution is found (TRUE), or not (FALSE).
cores	integer. Number of parallel cores to use in the machine to solve the problem.
verbose	logical. Indicates if the solver information is displayed while solving the optimization model (TRUE), or if it is not displayed (FALSE).
name_output_file	string. Prefix of all output names.
output_file	logical. Indicates if the outputs are exported as .csv files (TRUE), or they are not exported (FALSE). Currently, 5 files are exported. The distribution of actions in the solution, the distribution of the selected planning units, the benefits achieved by the features, the parameters used, and the optimization engine log.

Details

The solvers supported by the solve() function are described below.

Gurobi solver *Gurobi* is a state-of-the-art commercial optimization software with an R package interface. It is by far the fastest of the solvers available in this package, however, it is also the only solver that is not freely available. That said, licenses are available to academics at no cost. The **gurobi** package is distributed with the *Gurobi* software suite. This solver uses the **gurobi** package to solve problems.

Symphony solver *SYMPHONY* is an open-source integer programming solver that is part of the Computational Infrastructure for Operations Research (COIN-OR) project, an initiative to promote development of open-source tools for operations research (a field that includes linear programming). The **Rsymphony** package provides an interface to COIN-OR and is available on CRAN. This solver uses the **Rsymphony** package to solve problems.

Value

An object of class solution.

See Also

For more information on how to install and obtain an academic license of the Gurobi solver, see the *Gurobi installation guide*, which can be found online at prioritizr vignette.

Examples

```
## Not run:
## This example uses input files included into package.

## Load data
data(sim_pu_data, sim_features_data, sim_dist_features_data,
sim_threats_data, sim_dist_threats_data, sim_sensitivity_data,
sim_boundary_data)

## Create data instance
problem_data <- inputData(
  pu = sim_pu_data, features = sim_features_data, dist_features = sim_dist_features_data,
  threats = sim_threats_data, dist_threats = sim_dist_threats_data,
  sensitivity = sim_sensitivity_data, boundary = sim_boundary_data
)

## Create optimization model
problem_model <- problem(x = problem_data, blm = 1)

## Solve the optimization model using a gap_limit and gurobi solver
## NOTE: The Gurobi solver must be previously installed and must have a valid license!
s1 <- solve(a = problem_model, solver = "gurobi", gap_limit = 0.01, output_file = FALSE, cores = 2)

print(s1)

## Solve the optimization model using a gap_limit and symphony solver
s2 <- solve(a = problem_model,
            solver = "symphony",
            gap_limit = 0.01,
            output_file = FALSE,
            cores = 2)

print(s2)

## Solve the optimization model using a time_limit and gurobi solver
s3 <- solve(a = problem_model, solver = "gurobi", time_limit = 10, output_file = FALSE, cores = 2)

print(s3)

## End(Not run)
```

Index