Package ‘oppr’

August 11, 2019

Type Package

Version 0.0.4

Title Optimal Project Prioritization

Description A decision support tool for prioritizing conservation projects. Prioritizations can be developed by maximizing expected feature richness, expected phylogenetic diversity, the number of features that meet persistence targets, or identifying a set of projects that meet persistence targets for minimal cost. Constraints (e.g. lock in specific actions) and feature weights can also be specified to further customize prioritizations. After defining a project prioritization problem, solutions can be obtained using exact algorithms, heuristic algorithms, or random processes. In particular, it is recommended to install the 'Gurobi' optimizer (available from <https://www.gurobi.com>) because it can identify optimal solutions very quickly. Finally, methods are provided for comparing different prioritizations and evaluating their benefits.

Imports utils, methods, stats, Matrix, magrittr (>= 1.5), uuid (>= 0.1.2), proto (>= 1.0.0), cli (>= 1.0.1), assertthat (>= 0.2.0), tibble (>= 2.0.0), ape (>= 5.2), tidytree (>= 0.1.9), ggplot2 (>= 3.0.0), viridisLite (>= 0.3.0), lpSolveAPI (>= 5.5.2.0.17),

Suggests testthat (>= 2.0.0), knitr (>= 1.20), roxygen2 (>= 6.1.0), rmarkdown (>= 1.10), gurobi (>= 8.0.0), ggtree (>= 1.8.2), Rsymphony (>= 0.1.28), lpsymphony (>= 1.10.0), shiny (>= 1.2.0), rhandsontable (>= 0.3.7), tidyr (>= 0.8.2)

Depends R(>= 3.4.0)

LinkingTo Rcpp (>= 0.12.19), RcppArmadillo (>= 0.9.100.5.0), RcppProgress (>= 0.4.1)

License GPL-3

LazyData true

SystemRequirements C++11

URL https://prioritizr.github.io/oppr, https://github.com/prioritizr/oppr

BugReports https://github.com/prioritizr/oppr/issues

VignetteBuilder knitr

RoxygenNote 6.1.1

Collate 'internal.R' 'pproto.R' 'Parameter-proto.R'
'ArrayParameter-proto.R' 'MiscParameter-proto.R'
'Parameters-proto.R' 'ScalarParameter-proto.R' 'parameters.R'
'waiver.R' 'ProjectModifier-proto.R' 'Constraint-proto.R'
'Collection-proto.R' 'Decision-proto.R' 'Id.R'
'Objective-proto.R' 'OptimizationProblem-proto.R'
'OptimizationProblem-methods.R' 'ProjectProblem-proto.R'
'RcppExports.R' 'Solver-proto.R' 'Target-proto.R'
'Weight-proto.R' 'action_names.R' 'add_absolute_targets.R'
'add_binary_decisions.R' 'add_default_solver.R'
'add_feature_weights.R' 'add_gurobi_solver.R'
'add_heuristic_solver.R' 'add_locked_in_constraints.R'
'add_locked_out_constraints.R' 'add_lpsolveapi_solver.R'
'add_lpsymphony_solver.R' 'tbl_df.R' 'add_manual_targets.R'
'add_manual_locked_constraints.R'
'add_max_phylo_div_objective.R' 'star_phylogeny.R'
'add_max_richness_objective.R'
'add_max_targets_met_objective.R' 'add_min_set_objective.R'
'add_random_solver.R' 'add_relative_targets.R'
'add_rsymphony_solver.R' 'branch_matrix.R' 'compile.R'
'constraints.R' 'data.R' 'decisions.R' 'feature_names.R'
'magrittr-operators.R' 'misc.R' 'new_optimization_problem.R'
'number_of_actions.R' 'number_of_features.R'
'number_of_projects.R' 'objectives.R' 'package.R'
'solution_statistics.R' 'plot.R' 'plot_feature_persistence.R'
'plot_phylo_persistence.R' 'predefined_optimization_problem.R'
'print.R' 'problem.R' 'project_cost_effectiveness.R'
'project_names.R' 'rake_phylogeny.R' 'replacement_costs.R'
'show.R' 'simulate_ppp_data.R' 'simulate_ptm_data.R' 'solve.R'
'solvers.R' 'targets.R' 'weights.R' 'zzz.R'

NeedsCompilation yes

Author Jeffrey O Hanson [aut, cre],
Richard Schuster [aut],
Matthew Strimas-Mackey [aut],
Joseph Bennett [aut]

Maintainer Jeffrey O Hanson <jeffrey.hanson@uqconnect.edu.au>

Repository CRAN

Date/Publication 2019-08-10 22:30:07 UTC

R topics documented:

`action_names` *Action names*

Description

Extract the names of the actions in an object.

Usage

```
action_names(x)

## S4 method for signature 'ProjectProblem'
action_names(x)
```

Arguments

x	ProjectProblem-class.

Value

`character` action names.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with default solver
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
     add_binary_decisions() %>%
     add_default_solver()

# print problem
print(p)

# print action names
action_names(p)
```

`add_absolute_targets` *Add absolute targets*

Description

Set targets for a project prioritization `problem` by specifying exactly what probability of persistence is required for each feature. For instance, setting an absolute target of 10% (i.e. `0.1`) corresponds to a threshold 10% probability of persisting.

Usage

```
## S4 method for signature 'ProjectProblem,numeric'
add_absolute_targets(x, targets)

## S4 method for signature 'ProjectProblem,character'
add_absolute_targets(x, targets)
```

Arguments

`x`	`ProjectProblem-class` object.
`targets`	Object that specifies the targets for each feature. See the Details section for more information.

Details

Targets are used to specify the minimum probability of persistence for each feature in solutions. For minimum set objectives (i.e. `add_min_set_objective`, these targets specify the minimum probability of persistence required for each species in the solution. And for budget constrained objectives that use targets (i.e.`add_max_targets_met_objective`), these targets specify the minimum threshold probability of persistence that needs to be achieved to count the benefits for conserving these species. Please note that attempting to solve problems with objectives that require targets without specifying targets will throw an error.

The targets for a problem can be specified in several different ways:

`numeric vector` of target values for each feature. The order of the target values should correspond to the order of the features in the data used to create the argument to `x`. Additionally, for convenience, this type of argument can be a single value to assign the same target to each feature.

`character` specifying the name of column in the feature data (i.e. the argument to `features` in the `problem` function) that contains the persistence targets.

See Also

`targets`.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with minimum set objective and targets that require each
# feature to have a 30% chance of persisting into the future
p1 <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
      add_min_set_objective() %>%
      add_absolute_targets(0.3) %>%
      add_binary_decisions()

# print problem
print(p1)

# build problem with minimum set objective and specify targets that require
# different levels of persistence for each feature
p2 <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
      add_min_set_objective() %>%
      add_absolute_targets(c(0.1, 0.2, 0.3, 0.4, 0.5)) %>%
      add_binary_decisions()

# print problem
print(p2)

# add a column name to the feature data with targets
sim_features$target <- c(0.1, 0.2, 0.3, 0.4, 0.5)
```

```
# build problem with minimum set objective and specify targets using
# column name in the feature data
p3 <- problem(sim_projects, sim_actions, sim_features,
              "name", "success", "name", "cost", "name") %>%
      add_min_set_objective() %>%
      add_absolute_targets("target") %>%
      add_binary_decisions()

# print problem
print(p3)


# solve problems
s1 <- solve(p1)
s2 <- solve(p2)
s3 <- solve(p3)

# print solutions
print(s1)
print(s2)
print(s3)

# plot solutions
plot(p1, s1)
plot(p2, s2)
plot(p3, s3)
```

---

add_binary_decisions *Add binary decisions*

Description

Add a binary decision to a project prioritization problem. This is the conventional decision of either prioritizing funding for a management action or not.

Usage

```
add_binary_decisions(x)
```

Arguments

x	ProjectProblem-class object.

Details

Project prioritization problems involve making decisions about how funding will be allocated to management actions. Only a single decision should be added to a ProjectProblem object. If no decision is added to a problem then this decision type will be used by default. Currently, this is the only supported decision type.

Value

ProjectProblem-class object with the decisions added to it.

See Also

decisions.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with maximum richness objective, $200 budget, and
# binary decisions
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
     add_binary_decisions()

# print problem
print(p)


# solve problem
s <- solve(p)

# print solution
print(s)

# plot solution
plot(p, s)
```

---

add_default_solver *Add a default solver*

Description

Identify the best solver currently installed on the system and specify that it should be used to solve a project prioritization problem.

Usage

```
add_default_solver(x, ...)
```

Arguments

x	ProjectProblem-class object.
...	arguments passed to the solver.

Details

Ranked from best to worst, the solvers that can be used are: **gurobi**, (add_gurobi_solver), **Rsymphony** (add_rsymphony_solver), **lpsymphony** (add_lpsymphony_solver), and **lpSolveAPI** (add_lpsolveapi_solver). This function does not consider solvers that generate solutions using heuristic algorithms (i.e. add_heuristic_solver) or random processes (i.e. add_random_solver) because they cannot provide any guarantees on solution quality.

See Also

solvers.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with default solver
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
     add_binary_decisions() %>%
     add_default_solver()

# print problem
print(p)


# solve problem
s <- solve(p)

# print solution
print(s)

# plot solution
plot(p, s)
```

---

add_feature_weights *Add feature weights*

Description

Set weights for conserving features in a project prioritization problem.

Usage

```
## S4 method for signature 'ProjectProblem,numeric'
add_feature_weights(x, weights)
```

```
## S4 method for signature 'ProjectProblem,character'
add_feature_weights(x, weights)
```

Arguments

x ProjectProblem-class object.

weights Object that specifies the weights for each feature. See the Details section for more information.

Details

Weights are used to specify the relative importance for maintaining the persistence of specific features. For budget constrained problems (e.g. add_max_richness_objective), these weights could be used to specify which features are more important than other features according to evolutionary or cultural metrics. Specifically, features with a higher weight value are considered more important. It is generally best to ensure that the feature weights range between 0.0001 and 10,000 to reduce the time required to solve problems using exact algorithm solvers. As a consequence, you might have to rescale the feature weights if the largest or smallest values occur outside this range (excluding zeros). If you want to ensure that certain features conserved in the solutions, it is strongly recommended to lock in the actions for these features instead of setting really high weights for these features. Please note that a warning will be thrown if you attempt to solve problems with weights when an objective has been specified that does not use weights. Currently, all objectives—except for the minimum set objective (i.e. add_min_set_objective)—can use weights.

The weights for a problem can be specified in several different ways:

numeric vector of weight values for each feature.

character specifying the name of column in the feature data (i.e. the argument to features in the problem function) that contains the weights.

See Also

weights.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# print feature data
print(sim_features)

# build problem with maximum richness objective, $300 budget, and no weights
p1 <- problem(sim_projects, sim_actions, sim_features,
              "name", "success", "name", "cost", "name") %>%
      add_max_richness_objective(budget = 200) %>%
      add_binary_decisions()

# print problem
print(p1)
```

```
# build another problem, and specify feature weights using the values in the
# "weight" column of the sim_features table by specifying the column
# name "weight"
p2 <- p1 %>%
      add_feature_weights("weight")

# print problem
print(p2)

# build another problem, and specify feature weights using the
# values in the "weight column of the sim_features table, but
# actually input the values rather than specifying the column name
# "weights" column of the sim_features table
p3 <- p1 %>%
      add_feature_weights(sim_features$weight)

# print problem
print(p3)


# solve the problems
s1 <- solve(p1)
s2 <- solve(p2)
s3 <- solve(p3)

# print solutions
print(s1)
print(s2)
print(s3)

# plot solutions
plot(p1, s1)
plot(p2, s2)
plot(p3, s3)
```

---

add_gurobi_solver *Add a* **Gurobi** *solver*

Description

Specify that the *Gurobi* software should be used to solve a project prioritization `problem`. This function can also be used to customize the behavior of the solver. In addition to the *Gurobi* software suite, it also requires the **gurobi** package to be installed.

Usage

```
add_gurobi_solver(x, gap = 0, number_solutions = 1,
  solution_pool_method = 2, time_limit = .Machine$integer.max,
  presolve = 2, threads = 1, first_feasible = 0, verbose = TRUE)
```

Arguments

- `x` — `ProjectProblem-class` object.
- `gap` — `numeric` gap to optimality. This gap is relative when solving problems using **gurobi**, and will 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. For other solvers, this is the absolute gap, so if the optimal value for a maximization problem is 10, a gap of 0.01 means that a solution between 10 and 10.01 is required. Defaults to 0, so that optimal solutions will be returned.
- `number_solutions` — `integer` number of solutions desired. Defaults to 1. Note that the number of returned solutions can sometimes be less than the argument to `number_solutions` depending on the argument to `solution_pool_method`, for example if 100 solutions are requested but only 10 unique solutions exist, then only 10 solutions will be returned.
- `solution_pool_method` — `numeric` search method identifier that determines how multiple solutions should be generated. Available search modes for generating a portfolio of solutions include: `0` recording all solutions identified whilst trying to find a solution that is within the specified optimality gap, `1` finding one solution within the optimality gap and a number of additional solutions that are of any level of quality (such that the total number of solutions is equal to `number_solutions`), and `2` finding a specified number of solutions that are nearest to optimality. For more information, see the *Gurobi* manual (i.e. http://www.gurobi.com/documentation/8.0/refman/poolsearchmode.html#parameter:PoolSearchMode). Defaults to 2.
- `time_limit` — `numeric` time limit in seconds to run the optimizer. The solver will return the current best solution when this time limit is exceeded.
- `presolve` — `integer` number indicating how intensively the solver should try to simplify the problem before solving it. The default value of 2 indicates to that the solver should be very aggressive in trying to simplify the problem.
- `threads` — `integer` number of threads to use for the optimization algorithm. The default value of 1 will result in only one thread being used.
- `first_feasible` — `logical` should the first feasible solution be be returned? If `first_feasible` is set to `TRUE`, the solver will return the first solution it encounters that meets all the constraints, regardless of solution quality. Note that the first feasible solution is not an arbitrary solution, rather it is derived from the relaxed solution, and is therefore often reasonably close to optimality. Defaults to `FALSE`.
- `verbose` — `logical` should information be printed while solving optimization problems?

Details

Gurobi is a state-of-the-art commercial optimization software with an R package interface. It is by far the fastest of the solvers supported by 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.

To install the **gurobi** package, the Gurobi optimization suite will first need to be installed (see instructions for Linux, Mac OSX, and Windows operating systems). Although Gurobi is a commercial software, academics can obtain a special license for no cost. After installing the Gurobi optimization suite, the **gurobi** package can then be installed (see instructions for Linux, Mac OSX, and Windows operating systems).

Value

ProjectProblem-class object with the solver added to it.

See Also

solvers.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem
p1 <- problem(sim_projects, sim_actions, sim_features,
              "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
     add_binary_decisions()

# build another problem, and specify the Gurobi solver
p2 <- p1 %>%
      add_gurobi_solver()

# print problem
print(p2)

# solve problem
s2 <- solve(p2)

# print solution
print(s2)

# plot solution
plot(p2, s2)

# build another problem and obtain multiple solutions
# note that this problem doesn't have 100 unique solutions so
# the solver won't return 100 solutions
p3 <- p1 %>%
      add_gurobi_solver(number_solutions = 100)

# print problem
print(p3)

# solve problem
```

```
s3 <- solve(p3)

# print solutions
print(s3)
```

---

add_heuristic_solver *Add a heuristic solver*

Description

Specify that solutions should be generated using a backwards step-wise heuristic algorithm (inspired by Cabeza *et al.* 2004, Korte & Vygen 2000, Probert *et al.* 2016). Ideally, solutions should be generated using exact algorithm solvers (e.g. `add_rsymphony_solver` or `add_gurobi_solver`) because they are guaranteed to identify optimal solutions (Rodrigues & Gaston 2002).

Usage

```
add_heuristic_solver(x, number_solutions = 1, initial_sweep = TRUE,
  verbose = TRUE)
```

Arguments

`x`	`ProjectProblem-class` object.
`number_solutions`	`integer` number of solutions desired. Defaults to 1. Note that the number of returned solutions can sometimes be less than the argument to `number_solutions` depending on the argument to `solution_pool_method`, for example if 100 solutions are requested but only 10 unique solutions exist, then only 10 solutions will be returned.
`initial_sweep`	`logical` value indicating if projects and actions which exceed the budget should be automatically excluded prior to running the backwards heuristic. This step prevents projects which exceed the budget, and so would never be selected in the final solution, from biasing the cost-sharing calculations. However, previous algorithms for project prioritization have not used this step (e.g. Probert *et al.* 2016). Defaults to `TRUE`.
`verbose`	`logical` should information be printed while solving optimization problems?

Details

The heuristic algorithm used to generate solutions is described below. It is heavily inspired by the cost-sharing backwards heuristic algorithm conventionally used to guide the prioritization of species recovery projects (Probert *et al.* 2016).

1. All actions and projects are initially selected for funding.
2. A set of rules are then used to deselect actions and projects based on locked constraints (if any). Specifically, (i) actions which are which are locked out are deselected, and (ii) projects which are associated with actions that are locked out are also deselected.

3. If the argument to `initial_sweep` is `TRUE`, then a set of rules are then used to deselect actions and projects based on budgetary constraints (if present). Specifically, (i) actions which exceed the budget are deselected, (ii) projects which are associated with a set of actions that exceed the budget are deselected, and (iii) actions which are not associated with any project (excepting locked in actions) are also deselected.
4. If the objective function is to maximize biodiversity subject to budgetary constraints (e.g. `add_max_richness_objective`) then go to step 5. Otherwise, if the objective is to minimize cost subject to biodiversity constraints (i.e. `add_min_set_objective`) then go to step 7.
5. The next step is repeated until (i) the number of desired solutions is obtained, and (ii) the total cost of the remaining actions that are selected for funding is within the budget. After both of these conditions are met, the algorithm is terminated.
6. Each of the remaining projects that are currently selected for funding are evaluated according to how much the performance of the solution decreases when the project is deselected for funding, relative to the cost of the project not shared by other remaining projects. This can be expressed mathematically as:

$$B_j = \frac{V(J) - V(J - j)}{C_j}$$

   Where J is the set of remaining projects currently selected for funding (indexed by j), B_j is the benefit associated with funding project j, $V(J)$ is the objective value associated with the solution where all remaining projects are funded, $V(J - j)$ is the objective value associated with the solution where all remaining projects except for project j are funded, and C_j is the sum cost of all of the actions associated with project j—excluding locked in actions—with the cost of each action divided by the total number of remaining projects that share the action (e.g. if two projects both share a \$100 action, then this action contributes \$50 to the overall cost of each project).

   The project with the smallest benefit (i.e. B_j value) is then deselected for funding. In cases where multiple projects have the same benefit (B_j) value, the project with the greatest overall cost (including actions which are shared among multiple remaining projects) is deselected.
7. The next step is repeated until (i) the number of desired solutions is obtained or (ii) no action can be deselected for funding without the probability of any feature expecting to persist falling below its target probability of persistence. After one or both of these conditions are met, the algorithm is terminated.
8. Each of the remaining projects that are currently selected for funding are evaluated according to how much the performance of the solution decreases when the project is deselected for funding, relative to the cost of the project not shared by other remaining projects. This can be expressed mathematically as:

$$B_j = \frac{\left(\sum_f^F P_f(J) - T_f\right) - \left(\sum_f^F P_f(J - j) - T_f\right)}{C_j}$$

   Where F is the set of features (indexed by f), T_f is the target for feature f, P is the set of remaining projects that are selected for funding (indexed by j), C_j is the cost of all of the actions associated with project j—excluding locked in actions—and accounting for shared costs among remaining projects (e.g. if two projects both share a \$100 action, then this action contributes \$50 to the overall cost of each project), B_p is the benefit associated with funding project p, $P(J)$ is probability that each feature is expected to persist when the remaining projects (J) are funded, and $P(J - j)$ is the probability that each feature is expected to persist when all the remaining projects except for action j are funded.

The project with the smallest benefit (i.e. B_j value) is then deselected for funding. In cases where multiple projects have the same B_j value, the project with the greatest overall cost (including actions which are shared among multiple remaining projects) is deselected.

Value

`ProjectProblem-class` object with the solver added to it.

References

Rodrigues AS & Gaston KJ (2002) Optimisation in reserve selection procedures—why not? *Biological Conservation*, **107**, 123–129.

Cabeza M, Araujo MB, Wilson RJ, Thomas CD, Cowley MJ & Moilanen A (2004) Combining probabilities of occurrence with spatial reserve design. *Journal of Applied Ecology*, **41**, 252–262.

Korte B & Vygen J (2000) *Combinatorial Optimization. Theory and Algorithms*. Springer-Verlag, Berlin, Germany.

Probert W, Maloney RF, Mellish B, and Joseph L (2016) Project Prioritisation Protocol (PPP). Available at https://github.com/p-robot/ppp.

See Also

`solvers`.

Examples

```
# load ggplot2 package for making plots
library(ggplot2)

# load data
data(sim_projects, sim_features, sim_actions)

# build problem with heuristic solver and $200
p1 <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
     add_binary_decisions() %>%
     add_heuristic_solver()

# print problem
print(p1)


# solve problem
s1 <- solve(p1)

# print solution
print(s1)
```

```
# plot solution
plot(p1, s1)
```

`add_locked_in_constraints` *Add locked in constraints*

Description

Add constraints to a project prioritization `problem` to ensure that specific actions are prioritized for funding in the solution. For example, it may be desirable to lock in actions for conserving culturally or taxonomically important species.

Usage

```
add_locked_in_constraints(x, locked_in)

## S4 method for signature 'ProjectProblem,numeric'
add_locked_in_constraints(x, locked_in)

## S4 method for signature 'ProjectProblem,logical'
add_locked_in_constraints(x, locked_in)

## S4 method for signature 'ProjectProblem,character'
add_locked_in_constraints(x, locked_in)
```

Arguments

`x`	`ProjectProblem-class` object.
`locked_in`	Object that determines which planning units that should be locked in. See the Details section for more information.

Details

The locked actions can be specified in several different ways:

`integer vector` of indices pertaining to which actions should be locked in the solution (i.e. row numbers of the actions in the argument to `actions` in `problem`).

`logical vector` containing `logical` (i.e. `TRUE` and/or `FALSE` values) that indicate which actions should be locked in the solution. These `logical` values should correspond to each row in the argument to `actions` in `problem`).

`character` column name that indicates if actions units should be locked in the solution. This argument should denote a column in the argument to `actions` in `problem` which contains `logical` (i.e. `TRUE` and/or `FALSE` values) to indicate which actions should be locked.

Value

ProjectProblem-class object with the constraints added to it.

See Also

constraints.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# print action data
print(sim_actions)

# build problem with maximum richness objective and $150 budget
p1 <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 150) %>%
     add_binary_decisions()

# print problem
print(p1)

# build another problem, and lock in the 3rd action using numeric inputs
p2 <- p1 %>%
      add_locked_in_constraints(c(3))

# print problem
print(p2)

# build another problem, and lock in the actions using logical inputs from
# the sim_actions table
p3 <- p1 %>%
      add_locked_in_constraints(sim_actions$locked_in)

# print problem
print(p3)

# build another problem, and lock in the actions using the column name
# "locked_in" in the sim_actions table
# the sim_actions table
p4 <- p1 %>%
      add_locked_in_constraints("locked_in")

# print problem
print(p4)


# solve problems
s1 <- solve(p1)
s2 <- solve(p2)
```

```
s3 <- solve(p3)
s4 <- solve(p4)

# print the actions selected for funding in each of the solutions
print(s1[, sim_actions$name])
print(s2[, sim_actions$name])
print(s3[, sim_actions$name])
print(s4[, sim_actions$name])
```

---

add_locked_out_constraints
Add locked out constraints

Description

Add constraints to a project prioritization problem to ensure that specific actions are not prioritized for funding in the solution. For example, it may be desirable to lock out specific actions to examine their importance to the optimal funding scheme.

Usage

```
add_locked_out_constraints(x, locked_out)

## S4 method for signature 'ProjectProblem,numeric'
add_locked_out_constraints(x, locked_out)

## S4 method for signature 'ProjectProblem,logical'
add_locked_out_constraints(x, locked_out)

## S4 method for signature 'ProjectProblem,character'
add_locked_out_constraints(x, locked_out)
```

Arguments

x	ProjectProblem-class object.
locked_out	Object that determines which planning units that should be locked out. See the Details section for more information.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# update "locked_out" column to lock out action "F2_action"
sim_actions$locked_out <- c(FALSE, TRUE, FALSE, FALSE, FALSE, FALSE)

# print action data
```

```
print(sim_actions)

# build problem with maximum richness objective and $150 budget
p1 <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 150) %>%
     add_binary_decisions()

# print problem
print(p1)

# build another problem, and lock out the second action using numeric inputs
p2 <- p1 %>%
      add_locked_out_constraints(c(2))

# print problem
print(p2)

# build another problem, and lock out the actions using logical inputs
# (i.e. TRUE/FALSE values) from the sim_actions table
p3 <- p1 %>%
      add_locked_out_constraints(sim_actions$locked_out)

# print problem
print(p3)

# build another problem, and lock out the actions using the column name
# "locked_out" in the sim_actions table
# the sim_actions table
p4 <- p1 %>%
      add_locked_out_constraints("locked_out")

# print problem
print(p4)


# solve problems
s1 <- solve(p1)
s2 <- solve(p2)
s3 <- solve(p3)
s4 <- solve(p4)

# print the actions selected for funding in each of the solutions
print(s1[, sim_actions$name])
print(s2[, sim_actions$name])
print(s3[, sim_actions$name])
print(s4[, sim_actions$name])
```

add_lpsolveapi_solver   *Add a lp_solve solver with* **lpSolveAPI**

Description

Specify that the *lp_solve* software should be used to solve a project prioritization problem using the **lpSolveAPI** package. This function can also be used to customize the behavior of the solver. It requires the **lpSolveAPI** package.

Usage

```
add_lpsolveapi_solver(x, gap = 0, presolve = FALSE, verbose = TRUE)
```

Arguments

x	ProjectProblem-class object.
gap	numeric gap to optimality. This gap is relative when solving problems using **gurobi**, and will 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. For other solvers, this is the absolute gap, so if the optimal value for a maximization problem is 10, a gap of 0.01 means that a solution between 10 and 10.01 is required. Defaults to 0, so that optimal solutions will be returned.
presolve	logical indicating if attempts to should be made to simplify the optimization problem (TRUE) or not (FALSE). Defaults to TRUE.
verbose	logical should information be printed while solving optimization problems?

Details

lp_solve is an open-source integer programming solver. Although this solver is the slowest currently supported solver, it is also the only exact algorithm solver that can be installed on all operating systems without any manual installation steps. This solver is provided so that users can try solving small project prioritization problems, without needing to install additional software. When solve moderate or large project prioritization problems, consider using add_gurobi_solver.

Value

ProjectProblem-class object with the solver added to it.

See Also

solvers.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with lpSolveAPI solver
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
```

```
    add_binary_decisions() %>%
    add_lpsolveapi_solver()

# print problem
print(p)

# solve problem
s <- solve(p)

# print solution
print(s)

# plot solution
plot(p, s)
```

`add_lsymphony_solver` *Add a SYMPHONY solver with* **lpsymphony**

Description

Specify that the *SYMPHONY* software should be used to solve a project prioritization `problem` using the **lpsymphony** package. This function can also be used to customize the behavior of the solver. It requires the **lpsymphony** package.

Usage

```
add_lpsymphony_solver(x, gap = 0, time_limit = -1,
  first_feasible = 0, verbose = TRUE)
```

Arguments

`x`	`ProjectProblem-class` object.
`gap`	`numeric` gap to optimality. This gap is relative when solving problems using **gurobi**, and will 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. For other solvers, this is the absolute gap, so if the optimal value for a maximization problem is 10, a gap of 0.01 means that a solution between 10 and 10.01 is required. Defaults to 0, so that optimal solutions will be returned.
`time_limit`	`numeric` time limit in seconds to run the optimizer. The solver will return the current best solution when this time limit is exceeded.
`first_feasible`	`logical` should the first feasible solution be be returned? If `first_feasible` is set to `TRUE`, the solver will return the first solution it encounters that meets all the constraints, regardless of solution quality. Note that the first feasible solution is not an arbitrary solution, rather it is derived from the relaxed solution, and is therefore often reasonably close to optimality. Defaults to `FALSE`.
`verbose`	`logical` should information be printed while solving optimization problems?

Details

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 **lpsymphony** package is distributed through Bioconductor. This functionality is provided because the **lpsymphony** package may be easier to install to install on Windows and Mac OSX systems than the **Rsymphony** package.

Value

`ProjectProblem-class` object with the solver added to it.

See Also

`solvers`.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with lpsymphony solver
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
     add_binary_decisions() %>%
     add_lpsymphony_solver()

# print problem
print(p)

# solve problem
s <- solve(p)

# print solution
print(s)

# plot solution
plot(p, s)
```

`add_manual_locked_constraints`

Add manually specified locked constraints

Description

Add constraints to a project prioritization problem to ensure that solutions fund (or do not fund) specific actions. This function offers more fine-grained control than the add_locked_in_constraints and add_locked_out_constraints functions.

Usage

```
add_manual_locked_constraints(x, locked)

## S4 method for signature 'ProjectProblem,data.frame'
add_manual_locked_constraints(x, locked)

## S4 method for signature 'ProjectProblem,tbl_df'
add_manual_locked_constraints(x, locked)
```

Arguments

x	ProjectProblem-class object.
locked	data.frame or tibble object. See the Details section for more information.

Details

The argument to locked must contain the following fields (columns):

"action" character action name.

"status" numeric values indicating if actions should be funded (with a value of 1) or not (with a value of zero).

Value

ProjectProblem-class object with the constraints added to it.

See Also

constraints.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)


# create data frame with locked statuses
status <- data.frame(action = sim_actions$name[1:2],
                     status = c(0, 1))

# print locked statuses
print(status)

# build problem with minimum set objective and targets that require each
```

```
# feature to have a 30% chance of persisting into the future
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
      add_max_richness_objective(budget = 500) %>%
      add_manual_locked_constraints(status) %>%
      add_binary_decisions()

# print problem
print(p)


# solve problem
s <- solve(p)

# print solution
print(s)
```

---

add_manual_targets *Add manual targets*

Description

Set targets for a project prioritization problem by manually specifying all the required information for each target. This function is useful because it can be used to customize all aspects of a target. For most cases, targets can be specified using the add_absolute_targets and add_relative_targets functions. However, this function can be used to mix absolute and relative targets for different features.

Usage

```
## S4 method for signature 'ProjectProblem,data.frame'
add_manual_targets(x, targets)

## S4 method for signature 'ProjectProblem,tbl_df'
add_manual_targets(x, targets)
```

Arguments

x	ProjectProblem-class object.
targets	data.frame or tibble object. See the Details section for more information.

Details

Targets are used to specify the minimum probability of persistence for each feature in solutions. For minimum set objectives (i.e. add_min_set_objective, these targets specify the minimum probability of persistence required for each species in the solution. And for budget constrained objectives that use targets (i.e. add_max_targets_met_objective), these targets specify the minimum threshold probability of persistence that needs to be achieved to count the benefits for conserving these species. Please note that attempting to solve problems with objectives that require targets without specifying targets will throw an error.

The `targets` argument should contain the following columns:

`"feature"` `character` name of features in argument to `x`.

`"type"` `character` describing the type of target. Acceptable values include `"absolute"` and `"relative"`. These values correspond to `add_absolute_targets`, and `add_relative_targets` respectively.

`"sense"` `character` sense of the target. The only acceptable value currently supported is: `">="`. This field (column) is optional and if it is missing then target senses will default to `">="` values.

`"target"` `numeric` target threshold.

Value

`ProjectProblem-class` object with the targets added to it.

See Also

`targets`.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)


# create data frame with targets
targets <- data.frame(feature = sim_features$name,
                      type = "absolute",
                      target = 0.1)

# print targets
print(targets)

# build problem with minimum set objective and targets that require each
# feature to have a 30% chance of persisting into the future
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
      add_min_set_objective() %>%
      add_manual_targets(targets) %>%
      add_binary_decisions()

# print problem
print(p)


# solve problem
s <- solve(p)
```

```
# print solution
print(s)
```

---

add_max_phylo_div_objective

Add maximum phylogenetic diversity objective

Description

Set the objective of a project prioritization problem to maximize the phylogenetic diversity that is expected to persist into the future, whilst ensuring that the cost of the solution is within a pre-specified budget (Bennett *et al.* 2014, Faith 2008).

Usage

```
add_max_phylo_div_objective(x, budget, tree)
```

Arguments

x	ProjectProblem-class object.
budget	`numeric` budget for funding actions.
tree	phylo phylogenetic tree describing the evolutionary relationships between the features. Note that the argument to `tree` must contain every feature, and only the features, present in the argument to `x`.

Details

A problem objective is used to specify the overall goal of the project prioritization problem. Here, the maximum phylogenetic diversity objective seeks to find the set of actions that maximizes the expected amount of evolutionary history that is expected to persist into the future given the evolutionary relationships between the features (e.g. populations, species). Let I represent the set of conservation actions (indexed by i). Let C_i denote the cost for funding action i, and let m denote the maximum expenditure (i.e. the budget). Also, let F represent each feature (indexed by f), W_f represent the weight for each feature f (defaults to zero for each feature unless specified otherwise), and E_f denote the probability that each feature will go extinct given the funded conservation projects.

To describe the evolutionary relationships between the features $f \in F$, consider a phylogenetic tree that contains features $f \in F$ with branches of known lengths. This tree can be described using mathematical notation by letting B represent the branches (indexed by b) with lengths L_b and letting T_{bf} indicate which features $f \in F$ are associated with which phylogenetic branches $b \in B$ using zeros and ones. Ideally, the set of features F would contain all of the species in the study area—including non-threatened species—to fully account for the benefits for funding different actions.

To guide the prioritization, the conservation actions are organized into conservation projects. Let J denote the set of conservation projects (indexed by j), and let A_{ij} denote which actions $i \in I$ comprise each conservation project $j \in J$ using zeros and ones. Next, let P_j represent the probability of project j being successful if it is funded. Also, let B_{fj} denote the enhanced probability that each feature $f \in F$ associated with the project $j \in J$ will persist if all of the actions that comprise project j are funded and that project is allocated to feature f. For convenience, let Q_{fj} denote the actual probability that each $f \in F$ associated with the project $j \in J$ is expected to persist if the project is funded. If the argument to `adjust_for_baseline` in the `problem` function was set to `TRUE`, and this is the default behavior, then $Q_{fj} = (P_j \times B_{fj}) + \left(\left(1 - (P_j B_{fj})\right) \times (P_n \times B_{fn})\right)$, where `n` corresponds to the baseline "do nothing" project. This means that the probability of a feature persisting if a project is allocated to a feature depends on (i) the probability of the project succeeding, (ii) the probability of the feature persisting if the project does not fail, and (iii) the probability of the feature persisting even if the project fails. Otherwise, if the argument is set to `FALSE`, then $Q_{fj} = P_j \times B_{fj}$.

The binary control variables X_i in this problem indicate whether each project $i \in I$ is funded or not. The decision variables in this problem are the Y_j, Z_{fj}, E_f, and R_b variables. Specifically, the binary Y_j variables indicate if project j is funded or not based on which actions are funded; the binary Z_{fj} variables indicate if project j is used to manage feature f or not; the semi-continuous E_f variables denote the probability that feature f will go extinct; and the semi-continuous R_b variables denote the probability that phylogenetic branch b will remain in the future.

Now that we have defined all the data and variables, we can formulate the problem. For convenience, let the symbol used to denote each set also represent its cardinality (e.g. if there are ten features, let F represent the set of ten features and also the number ten).

$$\mathrm{Maximize}(\sum_{b=0}^{B} L_b R_b) + \sum_{f}^{F} (1 - E_f) W_f (\mathrm{eqn1a}) \mathrm{Subjectto} \sum_{i=0}^{I} C_i \leq m (\mathrm{eqn1b}) R_b = 1 - \prod_{f=0}^{F} ifelse(T_{bf} == 1, E_f, 1) \forall b \in$$

The objective (eqn 1a) is to maximize the expected phylogenetic diversity (Faith 2008) plus the probability each feature will remain multiplied by their weights (noting that the feature weights default to zero). Constraint (eqn 1b) limits the maximum expenditure (i.e. ensures that the cost of the funded actions do not exceed the budget). Constraints (eqn 1c) calculate the probability that each branch (including tips that correspond to a single feature) will go extinct according to the probability that the features which share a given branch will go extinct. Constraints (eqn 1d) calculate the probability that each feature will go extinct according to their allocated project. Constraints (eqn 1e) ensure that feature can only be allocated to projects that have all of their actions funded. Constraints (eqn 1f) state that each feature can only be allocated to a single project. Constraints (eqn 1g) ensure that a project cannot be funded unless all of its actions are funded. Constraints (eqns 1h) ensure that the probability variables (E_f) are bounded between zero and one. Constraints (eqns 1i) ensure that the action funding (X_i), project funding (Y_j), and project allocation (Z_{fj}) variables are binary.

Although this formulation is a mixed integer quadratically constrained programming problem (due to eqn 1c), it can be approximated using linear terms and then solved using commercial mixed integer programming solvers. This can be achieved by substituting the product of the feature extinction probabilities (eqn 1c) with the sum of the log feature extinction probabilities and using piecewise linear approximations (described in Hillier & Price 2005 pp. 390–392) to approximate the exponent of this term.

Value

`ProjectProblem-class` object with the objective added to it.

References

Bennett JR, Elliott G, Mellish B, Joseph LN, Tulloch AI, Probert WJ, Di Fonzo MMI, Monks JM, Possingham HP & Maloney R (2014) Balancing phylogenetic diversity and species numbers in conservation prioritization, using a case study of threatened species in New Zealand. *Biological Conservation*, **174**: 47–54.

Faith DP (2008) Threatened species and the potential loss of phylogenetic diversity: conservation scenarios based on estimated extinction probabilities and phylogenetic risk analysis. *Conservation Biology*, **22**: 1461–1470.

Hillier FS & Price CC (2005) *International series in operations research & management science*. Springer.

See Also

objectives.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions, sim_tree)

# plot tree
plot(sim_tree)

# build problem with maximum phylogenetic diversity objective and $300 budget
p1 <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
      add_max_phylo_div_objective(budget = 300, tree = sim_tree) %>%
      add_binary_decisions()


# solve problem
s1 <- solve(p1)

# print solution
print(s1)

# plot solution
plot(p1, s1)

# build another problem that includes feature weights
p2 <- p1 %>%
      add_feature_weights("weight")

# solve problem with feature weights
s2 <- solve(p2)

# print solution based on feature weights
print(s2)

# plot solution based on feature weights
plot(p2, s2)
```

add_max_richness_objective

Add maximum richness objective

Description

Set the objective of a project prioritization problem to maximize the total number of features that are expected to persist, whilst ensuring that the cost of the solution is within a pre-specified budget (Joseph, Maloney & Possingham 2009). This objective is conceptually similar to maximizing species richness in a study area. Furthermore, weights can also be used to specify the relative importance of conserving specific features (see add_feature_weights).

Usage

```
add_max_richness_objective(x, budget)
```

Arguments

x	ProjectProblem-class object.
budget	numeric budget for funding actions.

Details

A problem objective is used to specify the overall goal of the project prioritization problem. Here, the maximum richness objective seeks to find the set of actions that maximizes the total number of features (e.g. populations, species, ecosystems) that is expected to persist within a pre-specified budget. Let I represent the set of conservation actions (indexed by i). Let C_i denote the cost for funding action i, and let m denote the maximum expenditure (i.e. the budget). Also, let F represent each feature (indexed by f), W_f represent the weight for each feature f (defaults to one for each feature unless specified otherwise), and E_f denote the probability that each feature will go extinct given the funded conservation projects.

To guide the prioritization, the conservation actions are organized into conservation projects. Let J denote the set of conservation projects (indexed by j), and let A_{ij} denote which actions $i \in I$ comprise each conservation project $j \in J$ using zeros and ones. Next, let P_j represent the probability of project j being successful if it is funded. Also, let B_{fj} denote the probability that each feature $f \in F$ associated with the project $j \in J$ will persist if all of the actions that comprise project j are funded and that project is allocated to feature f. For convenience, let Q_{fj} denote the actual probability that each $f \in F$ associated with the project $j \in J$ is expected to persist if the project is funded. If the argument to adjust_for_baseline in the problem function was set to TRUE, and this is the default behavior, then $Q_{fj} = (P_j \times B_{fj}) + \Big(\big(1 - (P_j B_{fj})\big) \times (P_n \times B_{fn}) \Big)$, where n corresponds to the baseline "do nothing" project. This means that the probability of a feature persisting if a project is allocated to a feature depends on (i) the probability of the project succeeding, (ii) the probability of the feature persisting if the project does not fail, and (iii) the probability of the feature persisting even if the project fails. Otherwise, if the argument is set to `FALSE`, then $Q_{fj} = P_j \times B_{fj}$.

The binary control variables X_i in this problem indicate whether each project $i \in I$ is funded or not. The decision variables in this problem are the Y_j, Z_{fj}, and E_f variables. Specifically, the binary Y_j variables indicate if project j is funded or not based on which actions are funded; the binary Z_{fj} variables indicate if project j is used to manage feature f or not; and the semi-continuous E_f variables denote the probability that feature f will go extinct.

Now that we have defined all the data and variables, we can formulate the problem. For convenience, let the symbol used to denote each set also represent its cardinality (e.g. if there are ten features, let F represent the set of ten features and also the number ten).

$$\mathrm{Maximize} \sum_{f=0}^{F} (1 - E_f) W_f (\mathrm{eqn1a}) \mathrm{Subjectto} \sum_{i=0}^{I} C_i \leq m (\mathrm{eqn1b}) E_f = 1 - \sum_{j=0}^{J} Z_{fj} Q_{fj} \forall f \in F (\mathrm{eqn1c}) Z_{fj} \leq Y_j \forall j \in J (\mathrm{e}$$

The objective (eqn 1a) is to maximize the weighted persistence of all the species. Constraint (eqn 1b) limits the maximum expenditure (i.e. ensures that the cost of the funded actions do not exceed the budget). Constraints (eqn 1c) calculate the probability that each feature will go extinct according to their allocated project. Constraints (eqn 1d) ensure that feature can only be allocated to projects that have all of their actions funded. Constraints (eqn 1e) state that each feature can only be allocated to a single project. Constraints (eqn 1f) ensure that a project cannot be funded unless all of its actions are funded. Constraints (eqns 1g) ensure that the probability variables (E_f) are bounded between zero and one. Constraints (eqns 1h) ensure that the action funding (X_i), project funding (Y_j), and project allocation (Z_{fj}) variables are binary.

Value

`ProjectProblem-class` object with the objective added to it.

References

Joseph LN, Maloney RF & Possingham HP (2009) Optimal allocation of resources among threatened species: A project prioritization protocol. *Conservation Biology*, **23**, 328–338.

See Also

`objectives`.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with maximum richness objective and $300 budget
p1 <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
     add_binary_decisions()
```

```
# solve problem
s1 <- solve(p1)

# print solution
print(s1)

# plot solution
plot(p1, s1)


# build another problem that includes feature weights
p2 <- p1 %>%
     add_feature_weights("weight")


# solve problem with feature weights
s2 <- solve(p2)

# print solution based on feature weights
print(s2)

# plot solution based on feature weights
plot(p2, s2)
```

`add_max_targets_met_objective`
Add maximum targets met objective

Description

Set the objective of a project prioritization `problem` to maximize the total number of persistence targets met for the features, whilst ensuring that the cost of the solution is within a pre-specified budget (Chades *et al.* 2015). In some project prioritization exercises, decision makers may have a target level of persistence for each feature (e.g. a 90% persistence target corresponding to a 90% chance for the features persisting into the future). In such exercises, the decision makers do not perceive any benefit when a target is not met (e.g. if a feature has a persistence target of 90% and a solution only secures a 70% chance of persistence then no benefit is accrued for that feature) or when a target is surpassed (e.g. if a feature has a persistence target of 50%, then a solution which secures a 95% chance of persistence will accrue the same benefit as a solution which secures a 50% chance of persistence). Furthermore, weights can also be used to specify the relative importance of meeting targets for specific features (see `add_feature_weights`).

Usage

```
add_max_targets_met_objective(x, budget)
```

Arguments

x	ProjectProblem-class object.
budget	numeric budget for funding actions.

Details

A problem objective is used to specify the overall goal of the project prioritization problem. Here, the maximum targets met objective seeks to find the set of actions that maximizes the total number of features (e.g. populations, species, ecosystems) that have met their persistence targets within a pre-specified budget. Let I represent the set of conservation actions (indexed by i). Let C_i denote the cost for funding action i, and let m denote the maximum expenditure (i.e. the budget). Also, let F represent each feature (indexed by f), W_f represent the weight for each feature f (defaults to one for each feature unless specified otherwise), T_f represent the persistence target for each feature f, and E_f denote the probability that each feature will go extinct given the funded conservation projects.

To guide the prioritization, the conservation actions are organized into conservation projects. Let J denote the set of conservation projects (indexed by j), and let A_{ij} denote which actions $i \in I$ comprise each conservation project $j \in J$ using zeros and ones. Next, let P_j represent the probability of project j being successful if it is funded. Also, let B_{fj} denote the enhanced probability that each feature $f \in F$ associated with the project $j \in J$ will persist if all of the actions that comprise project j are funded and that project is allocated to feature f. For convenience, let Q_{fj} denote the actual probability that each $f \in F$ associated with the project $j \in J$ is expected to persist if the project is funded. If the argument to `adjust_for_baseline` in the `problem` function was set to `TRUE`, and this is the default behavior, then $Q_{fj} = (P_j \times B_{fj}) + \bigg(\big(1 - (P_j B_{fj})\big) \times (P_n \times B_{fn}) \bigg)$, where `n` corresponds to the baseline "do nothing" project. This means that the probability of a feature persisting if a project is allocated to a feature depends on (i) the probability of the project succeeding, (ii) the probability of the feature persisting if the project does not fail, and (iii) the probability of the feature persisting even if the project fails. Otherwise, if the argument is set to `FALSE`, then $Q_{fj} = P_j \times B_{fj}$.

The binary control variables X_i in this problem indicate whether each project $i \in I$ is funded or not. The decision variables in this problem are the Y_j, Z_{fj}, E_f, and G_f variables. Specifically, the binary Y_j variables indicate if project j is funded or not based on which actions are funded; the binary Z_{fj} variables indicate if project j is used to manage feature f or not; the semi-continuous E_f variables denote the probability that feature f will go extinct; and the G_f variables indicate if the persistence target for feature f is met.

Now that we have defined all the data and variables, we can formulate the problem. For convenience, let the symbol used to denote each set also represent its cardinality (e.g. if there are ten features, let F represent the set of ten features and also the number ten).

$$\mathrm{Maximize} \sum_{f=0}^{F} G_f W_f (\mathrm{eqn1a}) \mathrm{Subjectto} \sum_{i=0}^{I} C_i \leq m (\mathrm{eqn1b}) G_f (1 - E_f) \geq T_f \forall f \in F (\mathrm{eqn1c}) E_f = 1 - \sum_{j=0}^{J} Z_{fj} Q_{fj} \forall f \in I$$

The objective (eqn 1a) is to maximize the weighted total number of the features that have their persistence targets met. Constraints (eqn 1b) calculate which persistence targets have been met. Constraint (eqn 1c) limits the maximum expenditure (i.e. ensures that the cost of the funded actions do not exceed the budget). Constraints (eqn 1d) calculate the probability that each feature will go extinct according to their allocated project. Constraints (eqn 1e) ensure that feature can only be allocated to projects that have all of their actions funded. Constraints (eqn 1f) state that each feature can only be allocated to a single project. Constraints (eqn 1g) ensure that a project cannot be funded unless all of its actions are funded. Constraints (eqns 1h) ensure that the probability variables (E_f) are bounded between zero and one. Constraints (eqns 1i) ensure that the target met (G_f), action funding (X_i), project funding (Y_j), and project allocation (Z_{fj}) variables are binary.

Value

ProjectProblem-class object with the objective added to it.

References

Chades I, Nicol S, van Leeuwen S, Walters B, Firn J, Reeson A, Martin TG & Carwardine J (2015) Benefits of integrating complementarity into priority threat management. *Conservation Biology*, **29**, 525–536.

See Also

objectives.

Examples

```
# load the ggplot2 R package to customize plot
library(ggplot2)

# load data
data(sim_projects, sim_features, sim_actions)

# build problem with maximum targets met objective, a $200 budget,
# targets that require each feature to have a 20\% chance of persisting into
# the future, and zero cost actions locked in
p1 <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
      add_max_targets_met_objective(budget = 200) %>%
      add_absolute_targets(0.2) %>%
      add_locked_in_constraints(which(sim_actions$cost < 1e-5)) %>%
      add_binary_decisions()


# solve problem
s1 <- solve(p1)

# print solution
print(s1)

# plot solution, and add a dashed line to indicate the feature targets
# we can see the three features meet the targets under the baseline
# scenario, and the project for F3 was prioritized for funding
# so that its probability of persistence meets the target
plot(p1, s1) +
```

```
geom_hline(yintercept = 0.2, linetype = "dashed")


# build another problem that includes feature weights
p2 <- p1 %>%
      add_feature_weights("weight")


# solve problem
s2 <- solve(p2)

# print solution
print(s2)

# plot solution, and add a dashed line to indicate the feature targets
# we can see that adding weights to the problem has changed the solution
# specifically, the projects for feature F1 and F5 have been funded
# to enhance their probability of persistence
plot(p2, s2) +
geom_hline(yintercept = 0.2, linetype = "dashed")
```

---

add_min_set_objective *Add minimum set objective*

Description

Set the objective of a project prioritization problem to minimize the cost of the solution whilst ensuring that all targets are met. This objective is conceptually similar to that used in *Marxan* (Ball, Possingham & Watts 2009).

Usage

```
add_min_set_objective(x)
```

Arguments

x	ProjectProblem-class object.

Details

A problem objective is used to specify the overall goal of the project prioritization problem. Here, the minimum set objective seeks to find the set of actions that minimizes the overall cost of the prioritization, while ensuring that the funded projects meet a set of persistence targets for the conservation features (e.g. populations, species, ecosystems). Let I represent the set of conservation actions (indexed by i). Let C_i denote the cost for funding action i. Also, let F represent each feature (indexed by f), T_f represent the persistence target for feature f, and E_f denote the probability that each feature will go extinct given the funded conservation projects.

To guide the prioritization, the conservation actions are organized into conservation projects. Let J denote the set of conservation projects (indexed by j), and let A_{ij} denote which actions $i \in I$ comprise each conservation project $j \in J$ using zeros and ones. Next, let P_j represent the probability of project j being successful if it is funded. Also, let B_{fj} denote the enhanced probability that each feature $f \in F$ associated with the project $j \in J$ will persist if all of the actions that comprise project j are funded and that project is allocated to feature f. For convenience, let Q_{fj} denote the actual probability that each $f \in F$ associated with the project $j \in J$ is expected to persist if the project is funded. If the argument to `adjust_for_baseline` in the `problem` function was set to `TRUE`, and this is the default behavior, then $Q_{fj} = (P_j \times B_{fj}) + \left(\left(1 - (P_j B_{fj})\right) \times (P_n \times B_{fn}) \right)$, where `n` corresponds to the baseline "do nothing" project. This means that the probability of a feature persisting if a project is allocated to a feature depends on (i) the probability of the project succeeding, (ii) the probability of the feature persisting if the project does not fail, and (iii) the probability of the feature persisting even if the project fails. Otherwise, if the argument is set to `FALSE`, then $Q_{fj} = P_j \times B_{fj}$.

The binary control variables X_i in this problem indicate whether each project $i \in I$ is funded or not. The decision variables in this problem are the Y_j, Z_{fj}, and E_f variables. Specifically, the binary Y_j variables indicate if project j is funded or not based on which actions are funded; the binary Z_{fj} variables indicate if project j is used to manage feature f or not; and the semi-continuous E_f variables denote the probability that feature f will go extinct.

Now that we have defined all the data and variables, we can formulate the problem. For convenience, let the symbol used to denote each set also represent its cardinality (e.g. if there are ten features, let F represent the set of ten features and also the number ten).

$$\mathrm{Minimize} \sum_{i=0}^{I} C_i X_i (\mathrm{eqn1a}) \mathrm{Subjectto} (1 - E_f) \geq T_f \forall f \in F (\mathrm{eqn1b}) E_f = 1 - \sum_{j=0}^{J} Z_{fj} Q_{fj} \forall f \in F (\mathrm{eqn1c}) Z_{fj} \leq Y_j \forall j \in J$$

The objective (eqn 1a) is to minimize the cost of the funded actions. Constraints (eqn 1b) ensure that the persistence targets are met. Constraints (eqn 1c) calculate the probability that each feature will go extinct according to their allocated project. Constraints (eqn 1d) ensure that feature can only be allocated to projects that have all of their actions funded. Constraints (eqn 1e) state that each feature can only be allocated to a single project. Constraints (eqn 1f) ensure that a project cannot be funded unless all of its actions are funded. Constraints (eqns 1g) ensure that the probability variables (E_f) are bounded between zero and one. Constraints (eqns 1h) ensure that the action funding (X_i), project funding (Y_j), and project allocation (Z_{fj}) variables are binary.

Value

`ProjectProblem-class` object with the objective added to it.

References

Ball IR, Possingham HP & Watts M (2009) Marxan and relatives: software for spatial conservation prioritisation. *Spatial conservation prioritisation: Quantitative methods and computational tools*, 185-195.

See Also

`objectives`, `targets`.

Examples

```
# load the ggplot2 R package to customize plot
library(ggplot2)

# load data
data(sim_projects, sim_features, sim_actions)

# build problem with minimum set objective and targets that require each
# feature to have a 30\% chance of persisting into the future
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
      add_min_set_objective() %>%
      add_absolute_targets(0.3) %>%
      add_binary_decisions()


# solve problem
s <- solve(p)

# print solution
print(s)

# plot solution, and add a dashed line to indicate the feature targets
plot(p, s) +
geom_hline(yintercept = 0.3, linetype = "dashed")
```

`add_random_solver` *Add a random solver*

Description

Specify that solutions should be generated using random processes. Although prioritizations should be developed using optimization routines, a portfolio of randomly generated solutions can be useful for evaluating the effectiveness of solutions.

Usage

```
add_random_solver(x, number_solutions = 1, verbose = TRUE)
```

Arguments

`x` ProjectProblem-class object.

`number_solutions`
: `integer` number of solutions desired. Defaults to 1. Note that the number of returned solutions can sometimes be less than the argument to `number_solutions` depending on the argument to `solution_pool_method`, for example if 100 solutions are requested but only 10 unique solutions exist, then only 10 solutions will be returned.

`verbose` `logical` should information be printed while solving optimization problems?

Details

The algorithm used to randomly generate solutions depends on the the objective specified for the project prioritization problem.

For objectives which maximize benefit subject to budgetary constraints (e.g. add_max_richness_objective):

1. All locked in and zero-cost actions are initially selected for funding (excepting actions which are locked out).
2. A project—and all of its associated actions—is randomly selected for funding (excepting projects associated with locked out actions, and projects which would cause the budget to be exceeded when added to the existing set of selected actions).
3. The previous step is repeated until no more projects can be selected for funding without the total cost of the prioritized actions exceeding the budget.

For objectives which minimize cost subject to biodiversity constraints (i.e. add_min_set_objective:

1. All locked in and zero-cost actions are initially selected for funding (excepting actions which are locked out).
2. A project—and all of its associated actions—is randomly selected for funding (excepting projects associated with locked out actions, and projects which would cause the budget to be exceeded when added to the existing set of selected actions).
3. The previous step is repeated until all of the persistence targets are met.

Value

ProjectProblem-class object with the solver added to it.

See Also

solvers.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with random solver, and generate 100 random solutions
p1 <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
      add_max_richness_objective(budget = 200) %>%
      add_binary_decisions() %>%
      add_random_solver(number_solutions = 100)

# print problem
print(p1)

# solve problem
s1 <- solve(p1)

# print solutions
```

```
print(s1)

# plot first random solution
plot(p1, s1)

# plot histogram of the objective values for the random solutions
hist(s1$obj, xlab = "Expected richness", xlim = c(0, 2.5),
     main = "Histogram of random solutions")

# since the objective values don't tell us much about the quality of the
# solutions, we can find the optimal solution and calculate how different
# each of the random solutions is from optimality


# find the optimal objective value using an exact algorithms solver
s2 <- p1 %>%
      add_default_solver() %>%
      solve()

# create new column in s1 with percent difference from optimality
s1$optimality_diff <- ((s2$obj - s1$obj) / s1$obj) * 100

# plot histogram showing the quality of the random solutions
# higher numbers indicate worse solutions
hist(s1$optimality_diff, xlab = "Difference from optimality (%)",
     main = "Histogram of random solutions", xlim = c(0, 50))
```

---

add_relative_targets *Add relative targets*

Description

Set targets as a proportion (between 0 and 1) of the maximum probability of persistence associated with the best project for feature. For instance, if the best project for a feature has an 80 setting a 50% (i.e. `0.5`) relative target will correspond to a 40% threshold probability of persisting.

Usage

```
## S4 method for signature 'ProjectProblem,numeric'
add_relative_targets(x, targets)

## S4 method for signature 'ProjectProblem,character'
add_relative_targets(x, targets)
```

Arguments

`x`	`ProjectProblem-class` object.
`targets`	Object that specifies the targets for each feature. See the Details section for more information.

Details

Targets are used to specify the minimum probability of persistence for each feature in solutions. For minimum set objectives (i.e. add_min_set_objective, these targets specify the minimum probability of persistence required for each species in the solution. And for budget constrained objectives that use targets (i.e.add_max_targets_met_objective), these targets specify the minimum threshold probability of persistence that needs to be achieved to count the benefits for conserving these species. Please note that attempting to solve problems with objectives that require targets without specifying targets will throw an error.

The targets for a problem can be specified in several different ways:

numeric vector of target values for each feature. The order of the target values should correspond to the order of the features in the data used to create the argument to x. Additionally, for convenience, this type of argument can be a single value to assign the same target to each feature.

character specifying the name of column in the feature data (i.e. the argument to features in the problem function) that contains the persistence targets.

See Also

targets.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with minimum set objective and targets that require each
# feature to have a level of persistence that is greater than or equal to
# 70% of the best project for conserving it
p1 <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.7) %>%
      add_binary_decisions()

# print problem
print(p1)

# build problem with minimum set objective and specify targets that require
# different levels of persistence for each feature
p2 <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
      add_min_set_objective() %>%
      add_relative_targets(c(0.2, 0.3, 0.4, 0.5, 0.6)) %>%
      add_binary_decisions()

# print problem
print(p2)

# add a column name to the feature data with targets
```

```
sim_features$target <- c(0.2, 0.3, 0.4, 0.5, 0.6)

# build problem with minimum set objective and specify targets using
# column name in the feature data
p3 <- problem(sim_projects, sim_actions, sim_features,
              "name", "success", "name", "cost", "name") %>%
      add_min_set_objective() %>%
      add_relative_targets("target") %>%
      add_binary_decisions()


# print problem
print(p3)

# solve problems
s1 <- solve(p1)
s2 <- solve(p2)
s3 <- solve(p3)

# print solutions
print(s1)
print(s2)
print(s3)

# plot solutions
plot(p1, s1)
plot(p2, s2)
plot(p3, s3)
```

`add_rsymphony_solver` *Add a SYMPHONY solver with* **Rsymphony**

Description

Specify that the *SYMPHONY* software should be used to solve a project prioritization `problem` using the **Rsymphony** package. This function can also be used to customize the behavior of the solver. It requires the **Rsymphony** package.

Usage

```
add_rsymphony_solver(x, gap = 0, time_limit = -1, first_feasible = 0,
  verbose = TRUE)
```

Arguments

x	`ProjectProblem-class` object.

gap	numeric gap to optimality. This gap is relative when solving problems using **gurobi**, and will 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. For other solvers, this is the absolute gap, so if the optimal value for a maximization problem is 10, a gap of 0.01 means that a solution between 10 and 10.01 is required. Defaults to 0, so that optimal solutions will be returned.
time_limit	numeric time limit in seconds to run the optimizer. The solver will return the current best solution when this time limit is exceeded.
first_feasible	logical should the first feasible solution be be returned? If first_feasible is set to TRUE, the solver will return the first solution it encounters that meets all the constraints, regardless of solution quality. Note that the first feasible solution is not an arbitrary solution, rather it is derived from the relaxed solution, and is therefore often reasonably close to optimality. Defaults to FALSE.
verbose	logical should information be printed while solving optimization problems?

Details

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

ProjectProblem-class object with the solver added to it.

See Also

solvers.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with Rsymphony solver
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
     add_binary_decisions() %>%
     add_rsymphony_solver()

# print problem
print(p)

# solve problem
```

```
s <- solve(p)

# print solution
print(s)

# plot solution
plot(p, s)
```

ArrayParameter-class *Array parameter prototype*

Description

This prototype is used to represent a parameter has multiple values. Each value is has a label to differentiate values. **Only experts should interact directly with this prototype.**

Fields

$id `character` identifier for parameter.

$name `character` name of parameter.

$value `numeric` `vector` of values.

$label `character` `vector` of names for each value.

$default `numeric` `vector` of default values.

$length `integer` number of values.

$class `character` class of values.

$lower_limit `numeric` `vector` specifying the minimum permitted values.

$upper_limit `numeric` `vector` specifying the maximum permitted values.

$widget `function` used to construct a `shiny` interface for modifying values.

Usage

```
x$print()

x$show()

x$repr()

x$validate(tbl)

x$get()

x$set(tbl)

x$reset()

x$render(...)
```

Arguments

tbl data.frame containing new parameter values with row names indicating the labels and a column called "values" containing the new parameter values.

... arguments passed to function in `widget` field.

Details

print print the object.

show show the object.

repr `character` representation of object.

validate check if a proposed new set of parameters are valid.

get return a data.frame containing the parameter values.

set update the parameter values using a data.frame.

reset update the parameter values to be the default values.

render create a shiny widget to modify parameter values.

See Also

ScalarParameter-class, Parameter-class.

`array_parameters` *Array parameters*

Description

Create parameters that consist of multiple numbers. If an attempt is made to create a parameter with conflicting settings then an error will be thrown.

Usage

```
proportion_parameter_array(name, value, label)

binary_parameter_array(name, value, label)

integer_parameter_array(name, value, label,
  lower_limit = rep(as.integer(-.Machine$integer.max), length(value)),
  upper_limit = rep(as.integer(.Machine$integer.max), length(value)))

numeric_parameter_array(name, value, label,
  lower_limit = rep(.Machine$double.xmin, length(value)),
  upper_limit = rep(.Machine$double.xmax, length(value)))
```

Arguments

name	character name of parameter.
value	vector of values.
label	character vector of labels for each value.
lower_limit	vector of values denoting the minimum acceptable value for each element in value. Defaults to the smallest possible number on the system.
upper_limit	vector of values denoting the maximum acceptable value for each element in value. Defaults to the largest possible number on the system.

Details

Below is a list of parameter generating functions and a brief description of each.

proportion_parameter_array a parameter that consists of multiple numeric values that are between zero and one.

binary_parameter_array a parameter that consists of multiple integer values that are either zero or one.

integer_parameter_array a parameter that consists of multiple integer values.

numeric_parameter_array a parameter that consists of multiple numeric values.

Value

ArrayParameter-class object.

Examples

```
# proportion parameter array
p1 <- proportion_parameter_array('prop_array', c(0.1, 0.2, 0.3),
                                 letters[1:3])
print(p1) # print it
p1$get() # get value
p1$id # get id
invalid <- data.frame(value = 1:3, row.names=letters[1:3]) # invalid values
p1$validate(invalid) # check invalid input is invalid
valid <- data.frame(value = c(0.4, 0.5, 0.6), row.names=letters[1:3]) # valid
p1$validate(valid) # check valid input is valid
p1$set(valid) # change value to valid input
print(p1)

# binary parameter array
p2 <- binary_parameter_array('bin_array', c(0L, 1L, 0L), letters[1:3])
print(p2) # print it
p2$get() # get value
p2$id # get id
invalid <- data.frame(value = 1:3, row.names=letters[1:3]) # invalid values
p2$validate(invalid) # check invalid input is invalid
valid <- data.frame(value = c(0L, 0L, 0L), row.names=letters[1:3]) # valid
p2$validate(valid) # check valid input is valid
p2$set(valid) # change value to valid input
```

```
print(p2)

# integer parameter array
p3 <- integer_parameter_array('int_array', c(1:3), letters[1:3])
print(p3) # print it
p3$get() # get value
p3$id # get id
invalid <- data.frame(value = rnorm(3), row.names=letters[1:3]) # invalid
p3$validate(invalid) # check invalid input is invalid
valid <- data.frame(value = 5:7, row.names=letters[1:3]) # valid
p3$validate(valid) # check valid input is valid
p3$set(valid) # change value to valid input
print(p3)

# numeric parameter array
p4 <- numeric_parameter_array('dbl_array', c(0.1, 4, -5), letters[1:3])
print(p4) # print it
p4$get() # get value
p4$id # get id
invalid <- data.frame(value = c(NA, 1, 2), row.names=letters[1:3]) # invalid
p4$validate(invalid) # check invalid input is invalid
valid <- data.frame(value = c(1, 2, 3), row.names=letters[1:3]) # valid
p4$validate(valid) # check valid input is valid
p4$set(valid) # change value to valid input
print(p4)

# numeric parameter array with lower bounds
p5 <- numeric_parameter_array('b_dbl_array', c(0.1, 4, -5), letters[1:3],
                              lower_limit=c(0, 1, 2))
print(p5) # print it
p5$get() # get value
p5$id# get id
invalid <- data.frame(value = c(-1, 5, 5), row.names=letters[1:3]) # invalid
p5$validate(invalid) # check invalid input is invalid
valid <- data.frame(value = c(0, 1, 2), row.names=letters[1:3]) # valid
p5$validate(valid) # check valid input is valid
p5$set(valid) # change value to valid input
print(p5)
```

`as.Id` *Coerce object to another object*

Description

Coerce an object.

Usage

```
as.Id(x, ...)
```

```
## S3 method for class 'character'
as.Id(x, ...)

## S3 method for class 'Parameters'
as.list(x, ...)
```

Arguments

x	Object.
...	unused arguments.

Value

An Object.

---

as.list.OptimizationProblem

Convert `OptimizationProblem` *to list*

Description

Convert `OptimizationProblem` to list

Usage

```
## S3 method for class 'OptimizationProblem'
as.list(x, ...)
```

Arguments

x	`OptimizationProblem-class` object.
...	not used.

Value

`list` object.

`branch_matrix` *Branch matrix*

Description

Phylogenetic trees depict the evolutionary relationships between different species. Each branch in a phylogenetic tree represents a period of evolutionary history. Species that are connected to the same branch share the same period of evolutionary history represented by the branch. This function creates a matrix that shows which species are connected with which branches. In other words, it creates a matrix that shows which periods of evolutionary history each species has experienced.

Usage

```
branch_matrix(x, ...)

## Default S3 method:
branch_matrix(x, ...)

## S3 method for class 'phylo'
branch_matrix(x, assert_validity = TRUE, ...)
```

Arguments

- `x` — `phylo` tree object.
- `...` — not used.
- `assert_validity` — `logical` value (i.e. `TRUE` or `FALSE` indicating if the argument to `x` should be checked for validity. Defaults to `TRUE`.

Value

`dgCMatrix-class` sparse matrix object. Each row corresponds to a different species. Each column corresponds to a different branch. Species that inherit from a given branch are indicated with a one.

Examples

```
# load Matrix package to plot matrices
library(Matrix)

# load data
data(sim_tree)

# generate species by branch matrix
m <- branch_matrix(sim_tree)

# plot data
par(mfrow = c(1,2))
plot(sim_tree, main = "phylogeny")
```

```
image(m, main = "branch matrix")
```

`Collection-class` *Collection prototype*

Description

This prototype represents a collection of `ProjectModifier-class` objects.

Fields

$... `ProjectModifier-class` objects stored in the collection.

Usage

`x$print()`

`x$show()`

`x$repr()`

`x$ids()`

`x$length()`

`x$add`

`x$remove(id)`

`x$get_parameter(id)`

`x$set_parameter(id,value)`

`x$render_parameter(id)`

`x$render_all_parameters()`

Arguments

id `id` object.

value any object.

Details

print print the object.

show show the object.

repr `character` representation of object.

ids `character` ids for objects inside collection.

length `integer` number of objects inside collection.

add add `ProjectModifier-class` object.

remove remove an item from the collection.

get_parameter retrieve the value of a parameter in the object using an `id` object.

set_parameter change the value of a parameter in the object to a new object.

render_parameter generate a *shiny* widget to modify the the value of a parameter (specified by argument `id`).

render_all_parameters generate a `div` containing all the parameters" widgets.

See Also

`Constraint-class`.

`compile` *Compile a problem*

Description

Compile a project prioritization `problem` into a general purpose format for optimization.

Usage

```
compile(x, ...)

## S3 method for class 'ProjectProblem'
compile(x, ...)
```

Arguments

`x`	`ProjectProblem-class` object.
`...`	not used.

Details

This function might be useful for those interested in understanding how their project prioritization `problem` is expressed as a mathematical problem. However, if the problem just needs to be solved, then the `solve` function should be used instead.

Value

`OptimizationProblem-class` object.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with maximum richness objective, $200 budget, and
# binary decisions
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
     add_binary_decisions()

# print problem
print(p)

# compile problem
o <- compile(p)

# print compiled problem
print(o)
```

`Constraint-class` *Constraint prototype*

Description

This prototype is used to represent the constraints used when making a prioritization. **This prototype represents a recipe, to actually add constraints to a planning problem, see the help page on `constraints`. Only experts should use this class directly.** This prototype inherits from the `ProjectModifier-class`.

See Also

`ProjectModifier-class`.

`constraints` *Project prioritization problem constraints*

Description

A constraint can be added to a project prioritization `problem` to ensure that solutions exhibit a specific characteristic.

Details

The following constraints can be added to a project prioritization problem:

add_locked_in_constraints Add constraints to ensure that certain actions are prioritized for funding.

add_locked_out_constraints Add constraints to ensure that certain actions are not prioritized for funding.

See Also

decisions, objectives, problem, solvers, targets, weights.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with maximum richness objective and $150 budget
p1 <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 150) %>%
     add_binary_decisions()

# print problem
print(p1)

# build another problem, and lock in the third action
p2 <- p1 %>%
      add_locked_in_constraints(c(3))

# print problem
print(p2)

# build another problem, and lock out the second action
p3 <- p1 %>%
      add_locked_out_constraints(c(2))

# print problem
print(p3)


# solve problems
s1 <- solve(p1)
s2 <- solve(p2)
s3 <- solve(p3)

# print the actions selected for funding in each of the solutions
print(s1[, sim_actions$name])
print(s2[, sim_actions$name])
print(s3[, sim_actions$name])
```

Decision-class Decision prototype

Description

This prototype used to represent the type of decision that is made when prioritizing planning units. **This prototype represents a recipe to make a decision, to actually specify the type of decision in a planning problem, see the help page on decisions. Only experts should use this class directly.** This class inherits from the ProjectModifier-class.

See Also

ProjectModifier-class.

decisions Specify the type of decisions

Description

Project prioritization problems involve making decisions about how funding will be allocated to management actions.

Details

Please note that only one type of decision is currently supported:

add_binary_decisions This is the conventional type of decision where management actions are either prioritized for funding or not.

See Also

constraints, objectives, problem, solvers, targets, weights.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with maximum richness objective, $200 budget, and
# binary decisions
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
     add_binary_decisions()

# print problem
print(p)
```

```
# solve problem
s <- solve(p)

# print solution
print(s)

# plot solution
plot(p, s)
```

feature_names *Feature names*

Description

Extract the names of the features in an object.

Usage

```
feature_names(x)

## S4 method for signature 'ProjectProblem'
feature_names(x)
```

Arguments

x ProjectProblem-class.

Value

character feature names.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with default solver
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
     add_binary_decisions() %>%
     add_default_solver()

# print problem
print(p)

# print feature names
feature_names(p)
```

`is.Id` *Is it?*

Description

Test if an object inherits from a class.

Usage

```
is.Id(x)

is.Waiver(x)
```

Arguments

`x`	Object.

Value

`logical` indicating if it inherits from the class.

`matrix_parameters` *Matrix parameters*

Description

Create a parameter that represents a matrix object.

Usage

```
numeric_matrix_parameter(name, value, lower_limit = .Machine$double.xmin,
  upper_limit = .Machine$double.xmax, symmetric = FALSE)

binary_matrix_parameter(name, value, symmetric = FALSE)
```

Arguments

`name`	`character` name of parameter.
`value`	`matrix` object.
`lower_limit`	`numeric` values denoting the minimum acceptable value in the matrix. Defaults to the smallest possible number on the system.
`upper_limit`	`numeric` values denoting the maximum acceptable value in the matrix. Defaults to the smallest possible number on the system.
`symmetric`	`logical` must the must be matrix be symmetric? Defaults to `FALSE`.

Value

MiscParameter-class object.

Examples

```
# create matrix
m <- matrix(runif(9), ncol = 3)
colnames(m) <- letters[1:3]
rownames(m) <- letters[1:3]

# create a numeric matrix parameter
p1 <- numeric_matrix_parameter("m", m)
print(p1) # print it
p1$get() # get value
p1$id # get id
p1$validate(m[, -1]) # check if parameter can be updated
p1$set(m + 1) # set parameter to new values
p1$print() # print it again

# create a binary matrix parameter
m <- matrix(round(runif(9)), ncol = 3)
colnames(m) <- letters[1:3]
rownames(m) <- letters[1:3]

# create a binary matrix parameter
p2 <- binary_matrix_parameter("m", m)
print(p2) # print it
p2$get() # get value
p2$id # get id
p2$validate(m[, -1]) # check if parameter can be updated
p2$set(m + 1) # set parameter to new values
p2$print() # print it again
```

MiscParameter-class *Miscellaneous parameter prototype*

Description

This prototype is used to represent a parameter that can be any object. **Only experts should interact directly with this prototype.**

Fields

$id `character` identifier for parameter.

$name `character` name of parameter.

$value `tibble` object.

$validator `list` object containing a `function` that is used to validate changes to the parameter.

$widget `list` object containing a `function` used to construct a *shiny* interface for modifying values.

Usage

```
x$print()
x$show()
x$validate(x)
x$get()
x$set(x)
x$reset()
x$render(...)
```

Arguments

x object used to set a new parameter value.

... arguments passed to `$widget`.

Details

print print the object.

show show the object.

validate check if a proposed new parameter is valid.

get extract the parameter value.

set update the parameter value.

reset update the parameter value to be the default value.

render create a `shiny` widget to modify parameter values.

See Also

`Parameter-class`.

`misc_parameter` *Miscellaneous parameter*

Description

Create a parameter that consists of a miscellaneous object.

Usage

```
misc_parameter(name, value, validator, widget)
```

Arguments

name	character name of parameter.
value	object.
validator	function to validate changes to the parameter. This function must have a single argument and return either TRUE or FALSE depending on if the argument is valid candidate for the parameter.
widget	function to render a shiny widget. This function should must have a single argument that accepts a valid object and return a shiny.tag or shiny.tag.list object.

Value

MiscParameter-class object.

Examples

```
# load data
data(iris, mtcars)

# create table parameter can that can be updated to any other object
p1 <- misc_parameter("tbl", iris,
                     function(x) TRUE,
                     function(id, x) structure(id, .Class = "shiny.tag"))
print(p1) # print it
p1$get() # get value
p1$id # get id
p1$validate(mtcars) # check if parameter can be updated
p1$set(mtcars) # set parameter to mtcars
p1$print() # print it again

# create table parameter with validation function that requires
# all values in the first column to be less then 200 and that the
# parameter have the same column names as the iris data set
p2 <- misc_parameter("tbl2", iris,
                     function(x) all(names(x) %in% names(iris)) &&
                                 all(x[[1]] < 200),
                     function(id, x) structure(id, .Class = "shiny.tag"))
print(p2) # print it
p2$get() # get value
p2$id # get id
p2$validate(mtcars) # check if parameter can be updated
iris2 <- iris; iris2[1,1] <- 300 # create updated iris data set
p2$validate(iris2) # check if parameter can be updated
iris3 <- iris; iris2[1,1] <- 100 # create updated iris data set
p2$set(iris3) # set parameter to iris3
p2$print() # print it again
```

new_id — *Identifier*

Description

Generate a new unique identifier.

Usage

```
new_id()
```

Details

Identifiers are made using the `UUIDgenerate`.

Value

`Id` object.

See Also

`UUIDgenerate`.

Examples

```
# create new id
i <- new_id()

# print id
print(i)

# convert to character
as.character(i)

# check if it is an Id object
is.Id(i)
```

new_optimization_problem — *Optimization problem*

Description

Generate a new empty `OptimizationProblem-class` object.

Usage

```
new_optimization_problem()
```

Value

OptimizationProblem-class object.

See Also

OptimizationProblem-methods

Examples

```
# create empty OptimizationProblem object
x <- new_optimization_problem()

# print new object
print(x)
```

---

new_waiver *Waiver*

Description

Create a waiver object.

Usage

```
new_waiver()
```

Details

This object is used to represent that the user has not manually specified a setting, and so defaults should be used. By explicitly using a new_waiver(), this means that NULL objects can be a valid setting. The use of a "waiver" object was inspired by the ggplot2 package.

Value

object of class Waiver.

Examples

```
# create new waiver object
w <- new_waiver()

# print object
print(w)

# is it a waiver object?
```

```
is.Waiver(w)
```

---

number_of_actions *Number of actions*

Description

Extract the number of actions in an object.

Usage

```
number_of_actions(x)

## S4 method for signature 'ProjectProblem'
number_of_actions(x)

## S4 method for signature 'OptimizationProblem'
number_of_actions(x)
```

Arguments

x	ProjectProblem-class or OptimizationProblem-class object.

Value

integer number of actions.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with default solver
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
     add_binary_decisions() %>%
     add_default_solver()

# print problem
print(p)

# print number of actions
number_of_actions(p)
```

`number_of_features` *Number of features*

Description

Extract the number of features in an object.

Usage

```
number_of_features(x)

## S4 method for signature 'ProjectProblem'
number_of_features(x)

## S4 method for signature 'OptimizationProblem'
number_of_features(x)
```

Arguments

x	ProjectProblem-class or OptimizationProblem-class object.

Value

`integer` number of features.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with default solver
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
     add_binary_decisions() %>%
     add_default_solver()

# print problem
print(p)

# print number of features
number_of_features(p)
```

`number_of_projects` *Number of projects*

Description

Extract the number of projects in an object.

Usage

```
number_of_projects(x)

## S4 method for signature 'ProjectProblem'
number_of_projects(x)

## S4 method for signature 'OptimizationProblem'
number_of_projects(x)
```

Arguments

x	`ProjectProblem-class` or `OptimizationProblem-class` object.

Value

`integer` number of projects.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with default solver
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
     add_binary_decisions() %>%
     add_default_solver()

# print problem
print(p)

# print number of projects
number_of_projects(p)
```

`Objective-class` *Objective prototype*

Description

This prototype is used to represent an objective that can be added to a `ProjectProblem-class` object. **This prototype represents a recipe to make an objective, to actually add an objective to a planning problem: see `objectives`. Only experts should use this class directly.**

`objectives` *Problem objective*

Description

An objective is used to specify the overall goal of a project prioritization `problem`. All project prioritization problems involve minimizing or maximizing some kind of objective. For instance, the decision maker may require a funding scheme that maximizes the total number of species that are expected to persist into the future whilst ensuring that the total cost of the funded actions does not exceed a budget. Alternatively, the planner may require a solution that ensures that each species meets a target level of persistence whilst minimizing the cost of the funded actions. A project prioritization `problem` **must** have a specified objective before it can be solved, and attempting to solve a problem which does not have a specified objective will throw an error.

Details

The following objectives can be added to a conservation planning `problem`:

`add_max_richness_objective` Maximize the total number of features that are expected to persist, whilst ensuring that the cost of the solution is within a pre-specified budget (Joseph, Maloney & Possingham 2009).

`add_max_targets_met_objective` Maximize the total number of persistence targets met for the features, whilst ensuring that the cost of the solution is within a pre-specified budget (Chades *et al.* 2015).

`add_max_phylo_div_objective` Maximize the phylogenetic diversity that is expected to persist into the future, whilst ensuring that the cost of the solution is within a pre-specified budget (Bennett *et al.* 2014, Faith 2008).

`add_min_set_objective` Minimize the cost of the solution whilst ensuring that all targets are met. This objective is conceptually similar to that used in *Marxan* (Ball, Possingham & Watts 2009).

References

Ball IR, Possingham HP & Watts M (2009) Marxan and relatives: software for spatial conservation prioritisation. *Spatial conservation prioritisation: Quantitative methods and computational tools*, 185-195.

Bennett JR, Elliott G, Mellish B, Joseph LN, Tulloch AI, Probert WJ, Di Fonzo MMI, Monks JM, Possingham HP & Maloney R (2014) Balancing phylogenetic diversity and species numbers in conservation prioritization, using a case study of threatened species in New Zealand. *Biological Conservation*, **174**: 47–54.

Chades I, Nicol S, van Leeuwen S, Walters B, Firn J, Reeson A, Martin TG & Carwardine J (2015) Benefits of integrating complementarity into priority threat management. *Conservation Biology*, **29**, 525–536.

Faith DP (2008) Threatened species and the potential loss of phylogenetic diversity: conservation scenarios based on estimated extinction probabilities and phylogenetic risk analysis. *Conservation Biology*, **22**: 1461–1470.

Joseph LN, Maloney RF & Possingham HP (2009) Optimal allocation of resources among threatened species: A project prioritization protocol. *Conservation Biology*, **23**, 328–338.

See Also

constraints, decisions, problem, solvers, targets, weights.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions, sim_tree)

# build problem with maximum richness objective and $200 budget
p1 <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
     add_binary_decisions()


# solve problem
s1 <- solve(p1)

# print solution
print(s1)

# plot solution
plot(p1, s1)


# build problem with maximum phylogenetic diversity objective and $200 budget
p2 <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_phylo_div_objective(budget = 200, tree = sim_tree) %>%
     add_binary_decisions()
```

```
# solve problem
s2 <- solve(p2)

# print solution
print(s2)

# plot solution
plot(p2, s2)

# build problem with maximum targets met objective, $200 budget, and
# 40\% persistence targets
p3 <- problem(sim_projects, sim_actions, sim_features,
              "name", "success", "name", "cost", "name") %>%
     add_max_targets_met_objective(budget = 200) %>%
     add_absolute_targets(0.4) %>%
     add_binary_decisions()


# solve problem
s3 <- solve(p3)

# print solution
print(s3)

# plot solution
plot(p3, s3)


# build problem with minimum set objective, $200 budget, and 40\%
# persistence targets
p4 <- problem(sim_projects, sim_actions, sim_features,
              "name", "success", "name", "cost", "name") %>%
     add_min_set_objective() %>%
     add_absolute_targets(0.4) %>%
     add_binary_decisions()


# solve problem
s4 <- solve(p4)

# print solution
print(s4)

# plot solution
plot(p4, s4)
```

oppr **oppr**: *Optimal Project Prioritization*

Description

oppr: Optimal Project Prioritization

Description

The **oppr** *R* package a decision support tool for prioritizing conservation projects. Prioritizations can be developed by maximizing expected feature richness, expected phylogenetic diversity, the number of features that meet persistence targets, or identifying a set of projects that meet persistence targets for minimal cost. Constraints (e.g. lock in specific actions) and feature weights can also be specified to further customize prioritizations. After defining a project prioritization problem, solutions can be obtained using exact algorithms, heuristic algorithms, or random processes. In particular, it is recommended to install the 'Gurobi' optimizer (available from `https://www.gurobi.com`) because it can identify optimal solutions very quickly. Finally, methods are provided for comparing different prioritizations and evaluating their benefits.

Installation

To make the most of this package, the **ggtree** and **gurobi** R packages will need to be installed. Since the **ggtree** package is exclusively available at Bioconductor—and is not available on The Comprehensive R Archive Network—please execute the following command to install it: `source("https://bioconductor.org/bi` If the installation process fails, please consult the package's online documentation. To install the **gurobi** package, the Gurobi optimization suite will first need to be installed (see instructions for Linux, Mac OSX, and Windows operating systems). Although Gurobi is a commercial software, academics can obtain a special license for no cost. After installing the Gurobi optimization suite, the **gurobi** package can then be installed (see instructions for Linux, Mac OSX, and Windows operating systems).

See Also

Please refer to the package vignette for more information and worked examples. This can be accessed using the code `vignette("oppr")`.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# print project data
print(sim_projects)

# print action data
print(sim_features)

# print feature data
print(sim_actions)

# build problem
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 400) %>%
```

```
    add_feature_weights("weight") %>%
    add_binary_decisions()

# print problem
print(p)


# solve problem
s <- solve(p)

# print output
print(s)

# print which actions are funded in the solution
s[, sim_actions$name, drop = FALSE]

# print the expected probability of persistence for each feature
# if the solution were implemented
s[, sim_features$name, drop = FALSE]

# visualize solution
plot(p, s)
```

OptimizationProblem-class

Optimization problem class

Description

The `OptimizationProblem` class is used to represent an optimization problem. Data are stored in memory and accessed using an external pointer. **Only experts should interact with this class directly.**

Fields

$ptr `externalptr` object.

$data `list` object.

Usage

`x$print()`

`x$show()`

`x$repr()`

`x$ncol()`

`x$nrow()`

`x$ncell()`

```
x$modelsense()
x$vtype()
x$obj()
x$pwlobj()
x$A()
x$rhs()
x$sense()
x$lb()
x$ub()
x$number_of_projects()
x$number_of_actions()
x$number_of_features()
x$number_of_branches()
x$row_ids()
x$col_ids()
x$get_data()
```

Arguments

ptr `externalptr` object.

Details

print print the object.

show show the object.

repr `character` representation of object.

ncol `integer` number of columns (variables) in model matrix.

nrow `integer` number of rows (constraints) in model matrix.

ncell `integer` number of cells in model matrix.

modelsense `character` model sense.

vtype `character` vector of variable types.

obj `numeric` vector containing the linear components of the objective function.

pwlobj `list` object containing the piece-wise linear components of the objective function.

A `dgCMatrix-class` model matrix

rhs `numeric` vector of right-hand-side constraints.

sense `character` vector of constraint senses.

lb `numeric` vector of lower bounds for each decision variable.

ub `numeric` vector of upper bounds for each decision variable.

number_of_projects `integer` number of projects in the problem.

number_of_actions `integer` number of actions in the problem.

number_of_features `integer` number of features in the problem.

number_of_branches `integer` number of phylogenetic branches in the problem.

col_ids `character` names describing each decision variable (column) in the model matrix.

row_ids `character` names describing each constraint (row) in in the model matrix.

get_data `list` containing additional data.

`OptimizationProblem-methods` *Optimization problem methods*

Description

These functions are used to access data from an `OptimizationProblem-class` object.

Usage

```
nrow(x)

## S4 method for signature 'OptimizationProblem'
nrow(x)

ncol(x)

## S4 method for signature 'OptimizationProblem'
ncol(x)

ncell(x)

## S4 method for signature 'OptimizationProblem'
ncell(x)

modelsense(x)

## S4 method for signature 'OptimizationProblem'
modelsense(x)

vtype(x)

## S4 method for signature 'OptimizationProblem'
vtype(x)

obj(x)

## S4 method for signature 'OptimizationProblem'
```

```
obj(x)

pwlobj(x)

## S4 method for signature 'OptimizationProblem'
pwlobj(x)

A(x)

## S4 method for signature 'OptimizationProblem'
A(x)

rhs(x)

## S4 method for signature 'OptimizationProblem'
rhs(x)

sense(x)

## S4 method for signature 'OptimizationProblem'
sense(x)

lb(x)

## S4 method for signature 'OptimizationProblem'
lb(x)

ub(x)

## S4 method for signature 'OptimizationProblem'
ub(x)

col_ids(x)

## S4 method for signature 'OptimizationProblem'
col_ids(x)

row_ids(x)

## S4 method for signature 'OptimizationProblem'
row_ids(x)

number_of_branches(x)

## S4 method for signature 'OptimizationProblem'
number_of_branches(x)

get_data(x)
```

```
## S4 method for signature 'OptimizationProblem'
get_data(x)
```

Arguments

x OptimizationProblem-class object.

Details

The functions return the following data:

nrow integer number of rows (constraints).

ncol integer number of columns (decision variables).

ncell integer number of cells.

modelsense character describing if the problem is to be maximized ("max") or minimized ("min").

vtype character describing the type of each decision variable: binary ("B"), semi-continuous ("S"), or continuous ("C")

obj numeric vector defining the linear components of the objective function.

pwlobj list object defining the piece-wise linear components of the objective function.

A dgCMatrix-class matrix object defining the problem matrix.

rhs numeric vector with right-hand-side linear constraints

sense character vector with the senses of the linear constraints ("<=", ">=", "=").

lb numeric lower bound for each decision variable. Missing data values (NA) indicate no lower bound for a given variable.

ub numeric upper bounds for each decision variable. Missing data values (NA) indicate no upper bound for a given variable.

number_of_projects integer number of projects in the problem.

number_of_actions integer number of actions in the problem.

number_of_features integer number of features in the problem.

number_of_branches integer number of phylogenetic branches in the problem.

Value

list, dgCMatrix-class, numeric vector, numeric vector, or scalar integer depending on the method used.

`Parameter-class` *Parameter class*

Description

This class is used to represent a parameter that has multiple values. Each value has a different label to differentiate values. **Only experts should interact directly with this class.**

Fields

$id `Id` identifier for parameter.

$name `character` name of parameter.

$value `numeric` `vector` of values.

$default `numeric` `vector` of default values.

$class `character` name of the class that the values inherit from (e.g. `"integer"`.

$lower_limit `numeric` `vector` specifying the minimum permitted value for each element in `$value`.

$upper_limit `numeric` `vector` specifying the maximum permitted value for each element in `$value`.

$widget `function` used to construct a `shiny` interface for modifying values.

Usage

`x$print()`

`x$show()`

`x$reset()`

Details

print print the object.

show show the object.

reset change the parameter values to be the default values.

See Also

`ScalarParameter-class`.

parameters *Parameters*

Description

Create a new collection of Parameter objects.

Usage

```
parameters(...)
```

Arguments

...	Parameter-class objects.

Value

Parameters-class object.

See Also

array_parameters, scalar_parameters.

Examples

```
# create two Parameter objects
p1 <- binary_parameter("parameter one", 1)
print(p1)

p2 <- numeric_parameter("parameter two", 5)
print(p2)

# store Parameter objects in a Parameters object
p <- parameters(p1, p2)
print(p)
```

Parameters-class *Parameters class*

Description

This class represents a collection of Parameter-class objects. It provides methods for accessing, updating, and rendering the parameters stored inside it.

Fields

$parameters list object containing Parameter-class objects.

Usage

```
x$print()

x$show()

x$repr()

x$names()

x$ids()

x$length()

x$get(id)

x$set(id,value)

x$add(p)

x$render(id)

x$render_all()
```

Arguments

id `Id` object.

p `Parameter-class` object.

value any object.

Details

print print the object.

show show the object.

repr `character` representation of object.

names return `character` names of parameters.

ids return `character` parameter unique identifiers.

length return `integer` number of parameters in object.

get retrieve the value of a parameter in the object using an `Id` object.

set change the value of a parameter in the object to a new object.

render generate a *shiny* widget to modify the the value of a parameter (specified by argument `Id`).

render_all generate a `div` containing all the parameters" widgets.

plot.ProjectProblem *Plot a solution to a project prioritization problem*

Description

Create a plot to visualize how well a solution to a project prioritization problem will maintain biodiversity.

Usage

```
## S3 method for class 'ProjectProblem'
plot(x, solution, n = 1, symbol_hjust = 0.007,
  return_data = FALSE, ...)
```

Arguments

`x`	project prioritization `problem`.
`solution`	`data.frame` or `tibble` table containing the solutions. Here, rows correspond to different solutions and columns correspond to different actions. Each column in the argument to `solution` should be named according to a different action in `x`. Cell values indicate if an action is funded in a given solution or not, and should be either zero or one. Arguments to `solution` can contain additional columns, and they will be ignored.
`n`	`integer` solution number to visualize. Since each row in the argument to `solutions` corresponds to a different solution, this argument should correspond to a row in the argument to `solutions`. Defaults to 1.
`symbol_hjust`	`numeric` horizontal adjustment parameter to manually align the asterisks and dashes in the plot. Defaults to `0.007`. Increasing this parameter will shift the symbols further right. Please note that this parameter may require some tweaking to produce visually appealing publication quality plots.
`return_data`	`logical` should the underlying data used to create the plot be returned instead of the plot? Defaults to `FALSE`.
`...`	not used.

Details

The type of plot that this function creates depends on the problem objective. If the problem objective contains phylogenetic data, then this function plots a phylogenetic tree where each branch is colored according to its probability of persistence. Otherwise, if the problem does not contain phylogenetic data, then this function creates a bar plot where each bar corresponds to a different feature. The height of the bars indicate each feature's probability of persistence, and the width of the bars indicate each feature's weight.

Features that directly benefit from at least a single completely funded project with a non-zero cost are depicted with an asterisk symbol. Additionally, features that indirectly benefit from funded projects—because they are associated with partially funded projects that have non-zero costs and share actions with at least one funded project—are depicted with an open circle symbol.

Value

A ggplot object.

See Also

This function is essentially a wrapper for plot_feature_persistence and plot_phylo_persistence, so refer to the documentation for these functions for more information.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem without phylogenetic data
p1 <- problem(sim_projects, sim_actions, sim_features,
              "name", "success", "name", "cost", "name") %>%
      add_max_richness_objective(budget = 400) %>%
      add_feature_weights("weight") %>%
      add_binary_decisions()


# solve problem without phylogenetic data
s1 <- solve(p1)

# visualize solution without phylogenetic data
plot(p1, s1)


# build problem with phylogenetic data
p2 <- problem(sim_projects, sim_actions, sim_features,
              "name", "success", "name", "cost", "name") %>%
      add_max_phylo_div_objective(budget = 400, sim_tree) %>%
      add_binary_decisions()


# solve problem with phylogenetic data
s2 <- solve(p2)

# visualize solution with phylogenetic data
plot(p2, s2)
```

---

plot_feature_persistence

Plot a bar plot to visualize a project prioritization

Description

Create a bar plot to visualize how likely features are to persist into the future under a solution to a project prioritization problem.

Usage

```
plot_feature_persistence(x, solution, n = 1, symbol_hjust = 0.007,
  return_data = FALSE)
```

Arguments

x	project prioritization `problem`.
solution	`data.frame` or `tibble` table containing the solutions. Here, rows correspond to different solutions and columns correspond to different actions. Each column in the argument to `solution` should be named according to a different action in `x`. Cell values indicate if an action is funded in a given solution or not, and should be either zero or one. Arguments to `solution` can contain additional columns, and they will be ignored.
n	`integer` solution number to visualize. Since each row in the argument to `solutions` corresponds to a different solution, this argument should correspond to a row in the argument to `solutions`. Defaults to 1.
symbol_hjust	`numeric` horizontal adjustment parameter to manually align the asterisks and dashes in the plot. Defaults to `0.007`. Increasing this parameter will shift the symbols further right. Please note that this parameter may require some tweaking to produce visually appealing publication quality plots.
return_data	`logical` should the underlying data used to create the plot be returned instead of the plot? Defaults to `FALSE`.

Details

In this plot, each bar corresponds to a different feature. The length of each bar indicates the probability that a given feature will persist into the future, and the color of each bar indicates the weight for a given feature. Features that directly benefit from at least a single completely funded project with a non-zero cost are depicted with an asterisk symbol. Additionally, features that indirectly benefit from funded projects—because they are associated with partially funded projects that have non-zero costs and share actions with at least one completely funded project—are depicted with an open circle symbol.

Value

A `ggplot` object, or a `tbl_df` object if `return_data` is `TRUE`.

Examples

```
# set seed for reproducibility
set.seed(500)

# load the ggplot2 R package to customize plots
library(ggplot2)

# load data
data(sim_projects, sim_features, sim_actions)
```

```
# build problem
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
      add_max_richness_objective(budget = 400) %>%
      add_feature_weights("weight") %>%
      add_binary_decisions() %>%
      add_heuristic_solver(n = 10)


# solve problem
s <- solve(p)

# plot the first solution
plot(p, s)

# plot the second solution
plot(p, s, n = 2)

# since this function returns a ggplot2 plot object, we can customize the
# appearance of the plot using standard ggplot2 commands!
# for example, we can add a title
plot(p, s) + ggtitle("solution")

# we can also obtain the raw plotting data using return_data=TRUE
plot_data <- plot(p, s, return_data = TRUE)
print(plot_data)
```

`plot_phylo_persistence` *Plot a phylogram to visualize a project prioritization*

Description

Create a plot showing a phylogenetic tree (i.e. a "phylogram") to visualize the probability that phylogenetic branches are expected to persist into the future under a solution to a project prioritization problem.

Usage

```
plot_phylo_persistence(x, solution, n = 1, symbol_hjust = 0.007,
  return_data = FALSE)
```

Arguments

`x`	project prioritization `problem`.
`solution`	`data.frame` or `tibble` table containing the solutions. Here, rows correspond to different solutions and columns correspond to different actions. Each column in the argument to `solution` should be named according to a different action in `x`.

Cell values indicate if an action is funded in a given solution or not, and should be either zero or one. Arguments to `solution` can contain additional columns, and they will be ignored.

`n` — `integer` solution number to visualize. Since each row in the argument to `solutions` corresponds to a different solution, this argument should correspond to a row in the argument to `solutions`. Defaults to 1.

`symbol_hjust` — `numeric` horizontal adjustment parameter to manually align the asterisks and dashes in the plot. Defaults to `0.007`. Increasing this parameter will shift the symbols further right. Please note that this parameter may require some tweaking to produce visually appealing publication quality plots.

`return_data` — `logical` should the underlying data used to create the plot be returned instead of the plot? Defaults to `FALSE`.

Details

This function requires the **ggtree** (Yu *et al.* 2017). Since this package is distributed exclusively through Bioconductor, and is not available on the Comprehensive R Archive Network, please execute the following command to install it: `source("https://bioconductor.org/biocLite.R");biocLite("ggtree")`. If the installation process fails, please consult the package's online documentation.

In this plot, each phylogenetic branch is colored according to probability that it is expected to persist into the future (see Faith 2008). Features that directly benefit from at least a single completely funded project with a non-zero cost are depicted with an asterisk symbol. Additionally, features that indirectly benefit from funded projects—because they are associated with partially funded projects that have non-zero costs and share actions with at least one completely funded project—are depicted with an open circle symbol.

Value

A `ggtree` object, or a `treedata` object if `return_data` is `TRUE`.

References

Faith DP (2008) Threatened species and the potential loss of phylogenetic diversity: conservation scenarios based on estimated extinction probabilities and phylogenetic risk analysis. *Conservation Biology*, **22**: 1461–1470.

Yu G, Smith DK, Zhu H, Guan Y, & Lam TTY (2017) ggtree: an R package for visualization and annotation of phylogenetic trees with their covariates and other associated data. *Methods in Ecology and Evolution*, **8**: 28–36.

Examples

```
# set seed for reproducibility
set.seed(500)

# load the ggplot2 R package to customize plots
library(ggplot2)

data(sim_projects, sim_features, sim_actions)
```

```
# build problem
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_phylo_div_objective(budget = 400, sim_tree) %>%
     add_binary_decisions() %>%
     add_heuristic_solver(number_solutions = 10)


# solve problem
s <- solve(p)

# plot the first solution
plot(p, s)

# plot the second solution
plot(p, s, n = 2)

# since this function returns a ggplot2 plot object, we can customize the
# appearance of the plot using standard ggplot2 commands!
# for example, we can add a title
plot(p, s) + ggtitle("solution")

# we could also also set the minimum and maximum values in the color ramp to
# correspond to those in the data, rather than being capped at 0 and 1
plot(p, s) +
scale_color_gradientn(name = "Probability of\npersistence",
                      colors = viridisLite::inferno(150, begin = 0,
                                                    end = 0.9,
                                                    direction = -1)) +
ggtitle("solution")

# we could also change the color ramp
plot(p, s) +
scale_color_gradient(name = "Probability of\npersistence",
                     low = "red", high = "black") +
ggtitle("solution")

# we could even hide the legend if desired
plot(p, s) +
scale_color_gradient(name = "Probability of\npersistence",
                     low = "red", high = "black") +
theme(legend.position = "hide") +
ggtitle("solution")

# we can also obtain the raw plotting data using return_data=TRUE
plot_data <- plot(p, s, return_data = TRUE)
print(plot_data)
```

---

pproto *Create a new* pproto *object*

Description

Construct a new object with `pproto`. This object system is inspired from the `ggproto` system used in the `ggplot2` package.

Usage

```
pproto(`_class` = NULL, `_inherit` = NULL, ...)
```

Arguments

`_class`	Class name to assign to the object. This is stored as the class attribute of the object. This is optional: if `NULL` (the default), no class name will be added to the object.
`_inherit`	`pproto` object to inherit from. If `NULL`, don"t inherit from any object.
`...`	A list of members to add to the new `pproto` object.

Examples

```
Adder <- pproto("Adder",
  x = 0,
  add = function(self, n) {
    self$x <- self$x + n
    self$x
  }
)

Adder$add(10)
Adder$add(10)

Abacus <- pproto("Abacus", Adder,
  subtract = function(self, n) {
    self$x <- self$x - n
    self$x
  }
)
Abacus$add(10)
Abacus$subtract(10)
```

`print` *Print*

Description

Display information about an object.

Usage

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

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

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

## S4 method for signature 'Id'
print(x)

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

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

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

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

Arguments

x	Any object.
...	not used.

Value

None.

See Also

print.

Examples

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

problem *Project prioritization problem*

Description

Create a project prioritization problem. This function is used to specify the underlying data used in a prioritization problem: the projects, the management actions, and the features that need to be conserved (e.g. species, ecosystems). After constructing this `ProjectProblem-class` object, it can be customized using `objectives`, `targets`, `weights`, `constraints` and `solvers`. After building the problem, the `solve` function can be used to identify solutions.

Usage

```
problem(projects, actions, features, project_name_column,
  project_success_column, action_name_column, action_cost_column,
  feature_name_column, adjust_for_baseline = TRUE)
```

Arguments

`projects` `data.frame` or `tibble` table containing project data. Here, each row should correspond to a different project and columns should contain data that correspond to each project. This object should contain data that denote (i) the name of each project (specified in the argument to `project_name_column`), (ii) the probability that each project will succeed if all of its actions are funded (specified in the argument to `project_success_column`), (iii) the enhanced probability that each feature will persist if it is funded (using a column for each feature), and (iv) and which actions are associated with which projects (using a column for each action). This object must have a baseline project, with a zero cost value, that represents the probability that each feature will persist if no other conservation project is funded. Since each feature is assigned the greatest probability of persistence given the funded projects in a solution, the combined benefits of multiple projects can be encoded by creating additional projects that represent "combined projects". For instance, a habitat restoration project might cost \$100 and mean that a feature has a 40% chance of persisting, and a pest eradication project might cost \$50 and mean that a feature has a 60% chance of persisting. Due to non-linear effects, funding both of these projects might mean that a species has a 90% chance of persistence. This can be accounted for by creating a third project, representing the funding of both projects, which costs \$150 and provides a 90% chance of persistence.

`actions` `data.frame` or `tibble` table containing the action data. Here, each row should correspond to a different action and columns should contain data that correspond to each action. At a minimum, this object should contain data that denote (i) the name of each action (specified in the argument to `action_name_column`), (ii) the cost of each action (specified in the argument to `action_cost_column`). Optionally, it may also contain data that indicate actions should be (iii) locked in or (iv) locked out (see `add_locked_in_constraints` and `add_locked_out_constraints`).

It should also contain a zero-cost baseline action that is associated with the baseline project.

features
: data.frame or tibble table containing the feature data. Here, each row should correspond to a different feature and columns should contain data that correspond to each feature. At a minimum, this object should contain data that denote (i) the name of each feature (specified in the argument to feature_name_column). Optionally, it may also contain (ii) the weight for each feature or (iii) persistence targets for each feature.

project_name_column
: character name of column that contains the name for each conservation project. This argument corresponds to the projects table. Note that the project names must not contain any duplicates or missing values.

project_success_column
: character name of column that indicates the probability that each project will succeed. This argument corresponds to the argument to projects table. This column must have numeric values which range between zero and one. No missing values are permitted.

action_name_column
: character name of column that contains the name for each management action. This argument corresponds to the actions table. Note that the project names must not contain any duplicates or missing values.

action_cost_column
: character name of column that indicates the cost for funding each action. This argument corresponds to the argument to actions table. This column must have numeric values which are equal to or greater than zero. No missing values are permitted.

feature_name_column
: character name of the column that contains the name for each feature. This argument corresponds to the feature table. Note that the feature names must not contain any duplicates or missing values.

adjust_for_baseline
: logical should the probability of features persisting when projects are funded be adjusted to account for the probability of features persisting under the baseline "do nothing" scenario in the event that the funded projects fail to succeed? This should always be TRUE, except when funding a project means that the baseline "do nothing" scenario does not apply if a funded project fails. Defaults to TRUE.

Details

A project prioritization problem has actions, projects, and features. Features are the biological entities that need to be conserved (e.g. species, populations, ecosystems). Actions are real-world management actions that can be implemented to enhance biodiversity (e.g. habitat restoration, monitoring, pest eradication). Each action should have a known cost, and this usually means that each action should have a defined spatial extent and time period (though this is not necessary). Conservation projects are groups of management actions (they can also comprise a singular action too), and each project is associated with a probability of success if all of its associated actions are funded. To determine which projects should be funded, each project is associated with an probability of persistence for the features that they benefit. These values should indicate the probability that each feature will persist if only that project funded and not the additional benefit relative to the baseline project. Missing (`NA`) values should be used to indicate which projects do not enhance the probability of certain features.

The goal of a project prioritization exercise is then to identify which management actions—and as a consequence which conservation projects—should be funded. Broadly speaking, the goal of an optimization problem is to minimize (or maximize) an objective function given a set of control variables and decision variables that are subject to a series of constraints. In the context of project prioritization problems, the objective is usually some measure of utility (e.g. the net probability of each feature persisting into the future), the control variables determine which actions should be funded or not, the decision variables contain additional information needed to ensure correct calculations, and the constraints impose limits such as the total budget available for funding management actions. For more information on the mathematical formulations used in this package, please refer to the manual entries for the available objectives (listed in `objectives`).

Value

A fresh `ProjectProblem-class` object.

See Also

`constraints`, `decisions`, `objectives`, `solvers`, `targets`, `weights`, `solution_statistics`, `plot.ProjectProblem`.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# print project data
print(sim_projects)

# print action data
print(sim_features)

# print feature data
print(sim_actions)

# build problem
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 400) %>%
     add_feature_weights("weight") %>%
     add_binary_decisions()

# print problem
print(p)


# solve problem
```

```
s <- solve(p)

# print output
print(s)

# print which actions are funded in the solution
s[, sim_actions$name, drop = FALSE]

# print the expected probability of persistence for each feature
# if the solution were implemented
s[, sim_features$name, drop = FALSE]

# visualize solution
plot(p, s)
```

ProjectModifier-class *Conservation problem modifier prototype*

Description

This super-prototype is used to represent prototypes that in turn are used to modify a `ProjectProblem-class` object. Specifically, the `Constraint-class`, `Decision-class`, `Objective-class`, and `Target-class` prototypes inherit from this class. **Only experts should interact with this class directly because changes to these class will have profound and far reaching effects.**

Fields

$name `character` name of object.

$parameters `list` object used to customize the modifier.

$data `list` object with data.

$compressed_formulation `logical` can this constraint be applied to the compressed version of the conservation planning problem?. Defaults to `TRUE`.

Usage

`x$print()`

`x$show()`

`x$repr()`

`x$get_data(name)`

`x$set_data(name,value)`

`x$calculate(cp)`

`x$output()`

`x$apply(op,cp)`

`x$get_parameter(id)`

```
x$get_all_parameters()
x$set_parameter(id,value)
x$render_parameter(id)
x$render_all_parameter()
```

Arguments

name `character` name for object

value any object

id `id` or `name` of parameter

cp `ProjectProblem-class` object

op `OptimizationProblem-class` object

Details

print print the object.

show show the object.

repr return `character` representation of the object.

get_data return an object stored in the `data` field with the corresponding `name`. If the object is not present in the `data` field, a `waiver` object is returned.

set_data store an object stored in the `data` field with the corresponding name. If an object with that name already exists then the object is overwritten.

calculate `function` used to perform preliminary calculations and store the data so that they can be reused later without performing the same calculations multiple times. Data can be stored in the `data` slot of the input `ProjectModifier` or `ProjectProblem` objects.

output `function` used to generate an output from the object. This method is only used for `Target-class` objects.

apply `function` used to apply the modifier to an `OptimizationProblem-class` object. This is used by `Constraint-class`, `Decision-class`, and `Objective-class` objects.

get_parameter retrieve the value of a parameter.

get_all_parameters generate `list` containing all the parameters.

set_parameter change the value of a parameter to new value.

render_parameter generate a *shiny* widget to modify the the value of a parameter (specified by argument `id`).

render_all_parameters generate a `div` containing all the parameters" widgets.

ProjectProblem-class *Project problem class*

Description

Project problem class

Description

This class is used to represent project prioritization problems. A project prioritization problem has actions, projects, and features. Features are the biological entities that need to be conserved (e.g. species, populations, ecosystems). Actions are real-world management actions that can be implemented for conservation purposes (e.g. habitat restoration, monitoring, pest eradication). Each action should have a known cost, and this usually means that each action should have a defined spatial extent and time period (though this is not necessary). Conservation projects are groups of management actions (they can also comprise a singular action too), and each project is associated with a probability of success if all of its associated actions are funded. To determine which projects should be funded, each project is associated with an probability of persistence for the features that they benefit. These values should indicate the probability that each feature will persist if only that project funded and not the additional benefit relative to the baseline project. Missing (`NA`) values should be used to indicate which projects do not enhance the probability of certain features.

Given these data, a project prioritization problem involves making a decision about which actions should be funded or not—and in turn, which projects should be funded or not—to maximize or minimize a specific objective whilst meeting specific constraints. The objective for a project prioritization problem will *always* pertain to the probability that features are expected to persist. For example, an objective for a project prioritization problem could be to maximize the maximize the total amount of species that are expected to persist, or minimize the total cost of the funded actions subject to constraints which ensure that each feature meets a target level of persistence. The constraints in a project prioritization problem can be used to specify additional requirements (e.g. certain actions must be funded). Finally, a project prioritization problem—unlike an optimization problem—also requires a method to solve the problem. **This class represents a planning problem, to actually build and then solve a planning problem, use the `problem` function. Only experts should use this class directly.**

Fields

$data `list` object containing data.

$objective `Objective-class` object used to represent how the targets relate to the solution.

$decisions `Decision-class` object used to represent the type of decision made on planning units.

$targets `Target-class` object used to represent representation targets for features.

$weights `Weight-class` object used to represent feature weights.

$constraints `Collection-class` object used to represent additional `constraints` that the problem is subject to.

$solver `Solver-class` object used to solve the problem.

Usage

```
x$print()
x$show()
x$repr()
x$get_data(name)
x$set_data(name,value)
number_of_actions()
number_of_projects()
number_of_features()
action_names()
project_names()
feature_names()
feature_weights()
feature_phylogeny()
action_costs()
project_costs()
project_success_probabilities()
pf_matrix()
epf_matrix()
pa_matrix()
x$add_objective(obj)
x$add_decisions(dec)
x$add_constraint(con)
x$add_solver(sol)
x$add_targets(targ)
x$add_weights(wt)
x$get_constraint_parameter(id)
x$set_constraint_parameter(id,value)
x$render_constraint_parameter(id)
x$render_all_constraint_parameters()
x$get_objective_parameter(id)
x$set_objective_parameter(id,value)
x$render_objective_parameter(id)
x$render_all_objective_parameters()
x$get_solver_parameter(id)
x$set_solver_parameter(id,value)
x$render_solver_parameter(id)
x$render_all_solver_parameters()
```

Arguments

name character name for object.

value an object.

obj Objective-class object.

wt Weight-class object.

dec Decision-class object.

con Constraint-class object.

sol Solver-class object.

targ Target-class object.

wt Weight-class object.

id Id object that refers to a specific parameter.

value object that the parameter value should become.

Details

print print the object.

show show the object.

repr return character representation of the object.

get_data return an object stored in the data field with the corresponding name. If the object is not present in the data field, a waiver object is returned.

set_data store an object stored in the data field with the corresponding name. If an object with that name already exists then the object is overwritten.

number_of_actions integer number of actions.

number_of_projects integer number of projects.

number_of_features integer number of features.

action_names character names of actions in the problem.

project_names character names of projects in the problem.

feature_names character names of features in the problem.

feature_weights character feature weights.

feature_phylogeny phylo phylogenetic tree object.

action_costs numeric costs for each action.

project_costs numeric costs for each project.

project_success_probabilities numeric probability that each project will succeed.

pf_matrix dgCMatrix-class object denoting the enhanced probability that features will persist if different projects are funded.

epf_matrix dgCMatrix-class object denoting the enhanced probability that features is expected to persist if different projects are funded. This is calculated as the pf_matrix multiplied by the project success probabilities.

pa_matrix dgCMatrix-class object indicating which actions are associated with which projects.

feature_targets tibble with feature targets.

add_objective return a new ProjectProblem-class with the objective added to it.

add_decisions return a new ProjectProblem-class object with the decision added to it.

add_solver return a new ProjectProblem-class object with the solver added to it.

add_constraint return a new ProjectProblem-class object with the constraint added to it.

add_targets return a copy with the targets added to the problem.

get_constraint_parameter get the value of a parameter (specified by argument id) used in one of the constraints in the object.

set_constraint_parameter set the value of a parameter (specified by argument id) used in one of the constraints in the object to value.

render_constraint_parameter generate a *shiny* widget to modify the value of a parameter (specified by argument id).

render_all_constraint_parameters generate a *shiny* div containing all the parameters' widgets.

get_objective_parameter get the value of a parameter (specified by argument id) used in the object's objective.

set_objective_parameter set the value of a parameter (specified by argument id) used in the object's objective to value.

render_objective_parameter generate a *shiny* widget to modify the value of a parameter (specified by argument id).

render_all_objective_parameters generate a *shiny* div containing all the parameters' widgets.

get_weight_parameter get the value of a parameter (specified by argument id) used in the object's weights.

set_weight_parameter set the value of a parameter (specified by argument id) used in the object's weights to value.

render_weight_parameter generate a *shiny* widget to modify the value of a parameter (specified by argument id).

render_all_weight_parameters generate a *shiny* div containing all the parameters' widgets.

get_solver_parameter get the value of a parameter (specified by argument id) used in the object's solver.

set_solver_parameter set the value of a parameter (specified by argument id) used in the object's solver to value.

render_solver_parameter generate a *shiny* widget to modify the value of a parameter (specified by argument id).

render_all_solver_parameters generate a *shiny* div containing all the parameters' widgets.

`project_cost_effectiveness` *Project cost effectiveness*

Description

Calculate the individual cost-effectiveness of each conservation project in a project prioritization `problem` (Joseph, Maloney & Possingham 2009).

Usage

```
project_cost_effectiveness(x)
```

Arguments

`x`	project prioritization `problem`.

Details

Note that project cost-effectiveness cannot be calculated for problems with minimum set objectives because the objective function for these problems is to minimize cost and not maximize some measure of biodiversity persistence.

Value

A `tibble` table containing the following columns:

`"project"` `character` name of each project

`"cost"` `numeric` cost of each project.

`"benefit"` `numeric` benefit for each project. For a given project, this is calculated as the difference between (i) the objective value for a solution containing all of the management actions associated with the project and all zero cost actions, and (ii) the objective value for a solution containing the baseline project.

`"ce"` `numeric` cost-effectiveness of each project. For a given project, this is calculated as the difference between the the benefit for the project and the benefit for the baseline project, divided by the cost of the project. Note that the baseline project will have a `NaN` value because it has a zero cost.

`"rank"` `numeric` rank for each project according to is cost-effectiveness value. The project with a rank of one is the most cost-effective project. Ties are accommodated using averages.

References

Joseph LN, Maloney RF & Possingham HP (2009) Optimal allocation of resources among threatened species: A project prioritization protocol. *Conservation Biology*, **23**, 328–338.

See Also

solution_statistics, replacement_costs.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# print project data
print(sim_projects)

# print action data
print(sim_features)

# print feature data
print(sim_actions)

# build problem
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 400) %>%
     add_feature_weights("weight") %>%
     add_binary_decisions()

# print problem
print(p)

# calculate cost-effectiveness of each project
pce <- project_cost_effectiveness(p)

# print project costs, benefits, and cost-effectiveness values
print(pce)

# plot histogram of cost-effectiveness values
hist(pce$ce, xlab = "Cost effectiveness", main = "")
```

---

project_names *Project names*

Description

Extract the names of the projects in an object.

Usage

```
project_names(x)

## S4 method for signature 'ProjectProblem'
project_names(x)
```

Arguments

x ProjectProblem-class.

Value

character project names.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with default solver
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
     add_binary_decisions() %>%
     add_default_solver()

# print problem
print(p)

# print project names
project_names(p)
```

---

replacement_costs *Replacement cost*

Description

Calculate the replacement cost for priority actions in a project prioritization problem (Moilanen *et al.* 2009). Actions associated with larger replacement cost values are more irreplaceabe, and may need to be implemented sooner than actions with lower replacement cost values.

Usage

```
replacement_costs(x, solution, n = 1)
```

Arguments

x project prioritization problem.

solution data.frame or tibble table containing the solutions. Here, rows correspond to different solutions and columns correspond to different actions. Each column in the argument to solution should be named according to a different action in x. Cell values indicate if an action is funded in a given solution or not, and should be either zero or one. Arguments to solution can contain additional columns, and they will be ignored.

n integer solution number to calculate replacement cost values. Since each row in the argument to solutions corresponds to a different solution, this argument should correspond to a row in the argument to solutions. Defaults to 1.

Details

Replacement cost values are calculated for each priority action specified in the solution. Missing (NA) values are assigned to actions which are not selected for funding in the specified solution. For a given action, its replacement cost is calculated by (i) calculating the objective value for the optimal solution to the argument to x, (ii) calculating the objective value for the optimal solution to the argument to x with the given action locked out, (iii) calculating the difference between the two objective values, (iv) the problem has an objective which aims to minimize the objective value (only add_min_set_objective, then the resulting value is multiplied by minus one so that larger values always indicate actions with greater irreplaceability. Please note this function can take a long time to complete for large problems since it involves re-solving the problem for every action selected for funding.

Value

A tibble table containing the following columns:

"action" character name of each action.

"cost" numeric cost of each solution when each action is locked out.

"obj" numeric objective value of each solution when each action is locked out. This is calculated using the objective function defined for the argument to x.

"rep_cost" numeric replacement cost for each action. Greater values indicate greater irreplaceability. Missing (NA) values are assigned to actions which are not selected for funding in the specified solution, infinite (Inf) values are assigned to to actions which are required to meet feasibility constraints, and negative values mean that superior solutions than the specified solution exist.

References

Moilanen A, Arponen A, Stokland JN & Cabeza M (2009) Assessing replacement cost of conservation areas: how does habitat loss influence priorities? *Biological Conservation*, **142**, 575–585.

See Also

solution_statistics, project_cost_effectiveness.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with maximum richness objective and $400 budget
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 400) %>%
```

```
    add_feature_weights("weight") %>%
    add_binary_decisions()

# solve problem
s <- solve(p)

# print solution
print(s)

# calculate replacement cost values
r <- replacement_costs(p, s)

# print output
print(r)

# plot histogram of replacement costs,
# with this objective, greater values indicate greater irreplaceability
hist(r$rep_cost, xlab = "Replacement cost", main = "")
```

ScalarParameter-class *Scalar parameter prototype*

Description

This prototype is used to represent a parameter has a single value. **Only experts should interact directly with this prototype.**

Fields

$id `character` identifier for parameter.

$name `character` name of parameter.

$value `numeric` scalar value.

$default `numeric` scalar default value.

$class `character` name of the class that `$value` should inherit from (e.g. `integer`).

$lower_limit `numeric` scalar value that is the minimum value that `$value` is permitted to be.

$upper_limit `numeric` scalar value that is the maximum value that `$value` is permitted to be.

$widget `function` used to construct a `shiny` interface for modifying values.

Usage

`x$print()`

`x$show()`

`x$validate(x)`

`x$get()`

```
x$set(x)
x$reset()
x$render(...)
```

Arguments

x object used to set a new parameter value.

... arguments passed to `$widget`.

Details

print print the object.

show show the object.

validate check if a proposed new set of parameters are valid.

get extract the parameter value.

set update the parameter value.

reset update the parameter value to be the default value.

render create a `shiny` widget to modify parameter values.

See Also

`Parameter-class`, `ArrayParameter-class`.

---

scalar_parameters *Scalar parameters*

Description

These functions are used to create parameters that consist of a single number. Parameters have a name, a value, a defined range of acceptable values, a default value, a class, and a `shiny` widget for modifying them. If values are supplied to a parameter that are unacceptable then an error is thrown.

Usage

```
proportion_parameter(name, value)

binary_parameter(name, value)

integer_parameter(name, value,
  lower_limit = as.integer(-.Machine$integer.max),
  upper_limit = as.integer(.Machine$integer.max))

numeric_parameter(name, value, lower_limit = .Machine$double.xmin,
  upper_limit = .Machine$double.xmax)
```

Arguments

name	character name of parameter.
value	integer or double value depending on the parameter.
lower_limit	integer or double value representing the smallest acceptable value for value. Defaults to the smallest possible number on the system.
upper_limit	integer or double value representing the largest acceptable value for value. Defaults to the largest possible number on the system.

Details

Below is a list of parameter generating functions and a brief description of each.

proportion_parameter A parameter that is a double and bounded between zero and one.

integer_parameter A parameter that is a integer.

numeric_parameter A parameter that is a double.

binary_parameter A parameter that is restricted to integer values of zero or one.

Value

ScalarParameter-class object.

Examples

```
# proportion parameter
p1 <- proportion_parameter('prop', 0.5) # create new object
print(p1) # print it
p1$get() # get value
p1$id # get id
p1$validate(5) # check if 5 is a validate input
p1$validate(0.1) # check if 0.1 is a validate input
p1$set(0.1) # change value to 0.1
print(p1)

# binary parameter
p2 <- binary_parameter('bin', 0) # create new object
print(p2) # print it
p2$get() # get value
p2$id # get id
p2$validate(5) # check if 5 is a validate input
p2$validate(1L) # check if 1L is a validate input
p2$set(1L) # change value to 1L
print(p1) # print it again

# integer parameter
p3 <- integer_parameter('int', 5L) # create new object
print(p3) # print it
p3$get() # get value
p3$id # get id
p3$validate(5.6) # check if 5.6 is a validate input
```

```
p3$validate(2L) # check if 2L is a validate input
p3$set(2L) # change value to 2L
print(p3) # print it again

# numeric parameter
p4 <- numeric_parameter('dbl', -7.6) # create new object
print(p4) # print it
p4$get() # get value
p4$id # get id
p4$validate(NA) # check if NA is a validate input
p4$validate(8.9) # check if 8.9 is a validate input
p4$set(8.9) # change value to 8.9
print(p4) # print it again

# numeric parameter with lower bounds
p5 <- numeric_parameter('bdbl', 6, lower_limit=0) # create new object
print(p5) # print it
p5$get() # get value
p5$id # get id
p5$validate(-10) # check if -10 is a validate input
p5$validate(90) # check if 90 is a validate input
p5$set(90) # change value to 8.9
print(p5) # print it again
```

---

show *Show*

Description

Display information about an object.

Usage

```
## S4 method for signature 'ProjectModifier'
show(x)

## S4 method for signature 'ProjectProblem'
show(x)

## S4 method for signature 'Id'
show(x)

## S4 method for signature 'OptimizationProblem'
show(x)

## S4 method for signature 'Parameter'
show(x)
```

```
## S4 method for signature 'Solver'
show(x)
```

Arguments

`x`	Any object.
`...`	not used.

Value

None.

See Also

`show`.

`simulate_ppp_data` *Simulate data for the 'Project Prioritization Protocol'*

Description

Simulate data for developing project prioritizations. Here, data are simulated such that each feature has its own conservation project, similar to species-based prioritizations (e.g. Bennett *et al.* 2014).

Usage

```
simulate_ppp_data(number_features, cost_mean = 100, cost_sd = 5,
  success_min_probability = 0.7, success_max_probability = 0.99,
  funded_min_persistence_probability = 0.5,
  funded_max_persistence_probability = 0.9,
  baseline_min_persistence_probability = 0.01,
  baseline_max_persistence_probability = 0.4, locked_in_proportion = 0,
  locked_out_proportion = 0)
```

Arguments

`number_features`	`numeric` number of features.
`cost_mean`	`numeric` average cost for the actions. Defaults to `100`.
`cost_sd`	`numeric` standard deviation in action costs. Defaults to `5`.
`success_min_probability`	`numeric` minimum probability of the projects succeeding if they are funded. Defaults to `0.7`.
`success_max_probability`	`numeric` maximum probability of the projects succeeding if they are funded. Defaults to `0.99`.

funded_min_persistence_probability
: numeric minimum probability of the features persisting if projects are funded and successful. Defaults to 0.5.

funded_max_persistence_probability
: numeric maximum probability of the features persisting if projects are funded and successful. Defaults to 0.9.

baseline_min_persistence_probability
: numeric minimum probability of the features persisting if only the baseline project is funded. Defaults to 0.01.

baseline_max_persistence_probability
: numeric maximum probability of the features persisting if only the baseline project is funded. Defaults to 0.4.

locked_in_proportion
: numeric of actions that are locked into the solution. Defaults to 0.

locked_out_proportion
: numeric of actions that are locked into the solution. Defaults to 0.

Details

The simulated data set will contain one conservation project for each features, and also a "baseline" (do nothing) project to reflect features' persistence when their conservation project is not funded. Each conservation project is associated with a single action, and no conservation projects share any actions. Specifically, the data are simulated as follows:

1. A conservation project is created for each feature, and each project is associated with its own single action.
2. Cost data for each action are simulated using a normal distribution and the cost_mean and cost_sd arguments.
3. A set proportion of the actions are randomly set to be locked in and out of the solutions using the locked_in_proportion and locked_out_proportion arguments.
4. The probability of each project succeeding if its action is funded is simulated by drawing probabilities from a uniform distribution with the upper and lower bounds set as the success_min_probability and success_max_probability arguments.
5. The probability of each feature persisting if its project is funded and is successful is simulated by drawing probabilities from a uniform distribution with the upper and lower bounds set as the funded_min_persistence_probability and funded_max_persistence_probability arguments.
6. An additional project is created which represents the "baseline" (do nothing) scenario. The probability of each feature persisting when managed under this project is simulated by drawing probabilities from a uniform distribution with the upper and lower bounds set as the baseline_min_persistence_probability and baseline_max_persistence_probability arguments.
7. A phylogenetic tree is simulated for the features using rcoal.
8. Feature data are created from the phylogenetic tree. The weights are calculated as the amount of evolutionary history that has elapsed between each feature and its last common ancestor.

Value

A `list` object containing the elements:

`"projects"` A `tibble` containing the data for the conservation projects. It contains the following columns:

- `"name"` `character` name for each project.
- `"success"` `numeric` probability of each project succeeding if it is funded.
- `"F1"` **...** `"FN"` `numeric` columns for each feature, ranging from `"F1"` to `"FN"` where `N` is the number of features, indicating the enhanced probability that each feature will persist if it funded. Missing values (`NA`) indicate that a feature does not benefit from a project being funded.
- `"F1_action"` **...** `"FN_action"` `logical` columns for each action, ranging from `"F1_action"` to `"FN_action"` where `N` is the number of actions (equal to the number of features in this simulated data), indicating if an action is associated with a project (`TRUE`) or not (`FALSE`).
- `"baseline_action"` `logical` column indicating if a project is associated with the baseline action (`TRUE`) or not (`FALSE`). This action is only associated with the baseline project.

`"actions"` A `tibble` containing the data for the conservation actions. It contains the following columns:

- `"name"` `character` name for each action.
- `"cost"` `numeric` cost for each action.
- `"locked_in"` `logical` indicating if certain actions should be locked into the solution.
- `"locked_out"` `logical` indicating if certain actions should be locked out of the solution.

`"features"` A `tibble` containing the data for the conservation features (e.g. species). It contains the following columns:

- `"name"` `character` name for each feature.
- `"weight"` `numeric` weight for each feature. For each feature, this is calculated as the amount of time that elapsed between the present and the features' last common ancestor. In other words, the weights are calculated as the unique amount of evolutionary history that each feature has experienced.

"tree" `phylo` phylogenetic tree for the features.

References

Bennett JR, Elliott G, Mellish B, Joseph LN, Tulloch AI, Probert WJ, ... & Maloney R (2014) Balancing phylogenetic diversity and species numbers in conservation prioritization, using a case study of threatened species in New Zealand. *Biological Conservation*, **174**: 47–54.

See Also

`simulate_ptm_data`.

Examples

```
# create a simulated data set
s <- simulate_ppp_data(number_features = 5,
```

```
                       cost_mean = 100,
                       cost_sd = 5,
                       success_min_probability = 0.7,
                       success_max_probability = 0.99,
                       funded_min_persistence_probability = 0.5,
                       funded_max_persistence_probability = 0.9,
                       baseline_min_persistence_probability = 0.01,
                       baseline_max_persistence_probability = 0.4,
                       locked_in_proportion = 0.01,
                       locked_out_proportion = 0.01)

# print data set
print(s)
```

`simulate_ptm_data` *Simulate data for 'Priority threat management'*

Description

Simulate data for developing project prioritizations for a priority threat management exercise (Carwardine *et al.* 2019). Here, data are simulated for a pre-specified number of features, actions, and projects. Features can benefit from multiple projects, and different projects can share actions.

Usage

```
simulate_ptm_data(number_projects, number_actions, number_features,
  cost_mean = 100, cost_sd = 5, success_min_probability = 0.7,
  success_max_probability = 0.99,
  funded_min_persistence_probability = 0.5,
  funded_max_persistence_probability = 0.9,
  baseline_min_persistence_probability = 0.01,
  baseline_max_persistence_probability = 0.4, locked_in_proportion = 0,
  locked_out_proportion = 0)
```

Arguments

`number_projects`
: `numeric` number of projects. Note that this does not include the baseline project.

`number_actions`
: `numeric` number of actions. Note that this does not include the baseline action.

`number_features`
: `numeric` number of features.

`cost_mean`
: `numeric` average cost for the actions. Defaults to `100`.

`cost_sd`
: `numeric` standard deviation in action costs. Defaults to `5`.

`success_min_probability`
: `numeric` minimum probability of the projects succeeding if they are funded. Defaults to `0.7`.

success_max_probability
: numeric maximum probability of the projects succeeding if they are funded. Defaults to 0.99.

funded_min_persistence_probability
: numeric minimum probability of the features persisting if projects are funded and successful. Defaults to 0.5.

funded_max_persistence_probability
: numeric maximum probability of the features persisting if projects are funded and successful. Defaults to 0.9.

baseline_min_persistence_probability
: numeric minimum probability of the features persisting if only the baseline project is funded. Defaults to 0.01.

baseline_max_persistence_probability
: numeric maximum probability of the features persisting if only the baseline project is funded. Defaults to 0.4.

locked_in_proportion
: numeric of actions that are locked into the solution. Defaults to 0.

locked_out_proportion
: numeric of actions that are locked into the solution. Defaults to 0.

Details

The simulated data set will contain one conservation project for each features, and also a "baseline" (do nothing) project to reflect features' persistence when their conservation project is not funded. Each conservation project is associated with a single action, and no conservation projects share any actions. Specifically, the data are simulated as follows:

1. A specified number of conservation projects, features, and management actions are created.
2. Cost data for each action are simulated using a normal distribution and the `cost_mean` and `cost_sd` arguments.
3. A set proportion of the actions are randomly set to be locked in and out of the solutions using the `locked_in_proportion` and `locked_out_proportion` arguments.
4. The probability of each project succeeding if its action is funded is simulated by drawing probabilities from a uniform distribution with the upper and lower bounds set as the `success_min_probability` and `success_max_probability` arguments.
5. The probability of each feature persisting if various projects are funded and is successful is simulated by drawing probabilities from a uniform distribution with the upper and lower bounds set as the `funded_min_persistence_probability` and `funded_max_persistence_probability` arguments. To prevent
6. An additional project is created which represents the "baseline" (do nothing) scenario. The probability of each feature persisting when managed under this project is simulated by drawing probabilities from a uniform distribution with the upper and lower bounds set as the `baseline_min_persistence_probability` and `baseline_max_persistence_probability` arguments.
7. A phylogenetic tree is simulated for the features using `rcoal`.
8. Feature data are created from the phylogenetic tree. The weights are calculated as the amount of evolutionary history that has elapsed between each feature and its last common ancestor.

Value

A `list` object containing the elements:

`"projects"` A `tibble` containing the data for the conservation projects. It contains the following columns:

- `"name"` `character` name for each project.
- `"success"` `numeric` probability of each project succeeding if it is funded.
- `"F1"` **...** `"FN"` `numeric` columns for each feature, ranging from `"F1"` to `"FN"` where `N` is the number of features, indicating the enhanced probability that each feature will persist if it funded. Missing values (`NA`) indicate that a feature does not benefit from a project being funded.
- `"F1_action"` **...** `"FN_action"` `logical` columns for each action, ranging from `"F1_action"` to `"FN_action"` where `N` is the number of actions (equal to the number of features in this simulated data), indicating if an action is associated with a project (`TRUE`) or not (`FALSE`).
- `"baseline_action"` `logical` column indicating if a project is associated with the baseline action (`TRUE`) or not (`FALSE`). This action is only associated with the baseline project.

`"actions"` A `tibble` containing the data for the conservation actions. It contains the following columns:

- `"name"` `character` name for each action.
- `"cost"` `numeric` cost for each action.
- `"locked_in"` `logical` indicating if certain actions should be locked into the solution.
- `"locked_out"` `logical` indicating if certain actions should be locked out of the solution.

`"features"` A `tibble` containing the data for the conservation features (e.g. species). It contains the following columns:

- `"name"` `character` name for each feature.
- `"weight"` `numeric` weight for each feature. For each feature, this is calculated as the amount of time that elapsed between the present and the features' last common ancestor. In other words, the weights are calculated as the unique amount of evolutionary history that each feature has experienced.

"tree" `phylo` phylogenetic tree for the features.

References

Carwardine J, Martin TG, Firn J, Ponce-Reyes P, Nicol S, Reeson A, Grantham HS, Stratford D, Kehoe L, Chades I (2019) Priority Threat Management for biodiversity conservation: A handbook. *Journal of Applied Ecology*, **In press**: https://doi.org/10.1111/1365-2664.13268.

See Also

`simulate_ppp_data`.

Examples

```
# create a simulated data set
s <- simulate_ptm_data(number_projects = 6,
```

```
                           number_actions = 8,
                           number_features = 5,
                           cost_mean = 100,
                           cost_sd = 5,
                           success_min_probability = 0.7,
                           success_max_probability = 0.99,
                           funded_min_persistence_probability = 0.5,
                           funded_max_persistence_probability = 0.9,
                           baseline_min_persistence_probability = 0.01,
                           baseline_max_persistence_probability = 0.4,
                           locked_in_proportion = 0.01,
                           locked_out_proportion = 0.01)

# print data set
print(s)
```

`sim_data` *Simulated data*

Description

Simulated data for prioritizing conservations projects.

Usage

```
data(sim_projects)

data(sim_actions)

data(sim_features)

data(sim_tree)
```

Format

sim_projects `tibble` object.

sim_actions `tibble` object.

sim_features `tibble` object.

sim_tree `phylo` object.

Details

The data set contains the following objects:

`sim_projects` A `tibble` containing data for six simulated conservation projects. Each row corresponds to a different project and each column contains information about the projects. This table contains the following columns:

"name" character name for each project.

"success" numeric probability of each project succeeding if it is funded.

"F1" ... "F5" numeric columns for each feature (i.e. "F1", "F2", "F3", "F4", "F5", indicating the enhanced probability that each feature will survive if it funded. Missing values (NA) indicate that a feature does not benefit from a project being funded.

"F1_action" ... "F5_action" logical columns for each action, ranging from "F1_action" to "F5_action" indicating if an action is associated with a project (TRUE) or not (FALSE).

"baseline_action" logical column indicating if a project is associated with the baseline action (TRUE) or not (FALSE). This action is only associated with the baseline project.

sim_actions A tibble containing data for six simulated actions. Each row corresponds to a different action and each column contains information about the actions. This table contains the following columns:

"name" character name for each action.

"cost" numeric cost for each action.

"locked_in" logical indicating if certain actions should be locked into the solution.

"locked_out" logical indicating if certain actions should be locked out of the solution.

sim_features A tibble containing data for five simulated features. Each row corresponds to a different feature and each column contains information about the features. This table contains the following columns:

"name" character name for each feature.

"weight" numeric weight for each feature.

tree phylo phylogenetic tree for the features.

Examples

```
# load data
data(sim_projects, sim_actions, sim_features, sim_tree)

# print project data
print(sim_projects)
# print action data
print(sim_actions)

# print feature data
print(sim_features)
# plot phylogenetic tree
plot(sim_tree)
```

---

solution_statistics *Solution statistics*

Description

Calculate statistics describing a solution to a project prioritization problem.

Usage

```
solution_statistics(x, solution)
```

Arguments

x	project prioritization `problem`.
solution	`data.frame` or `tibble` table containing the solutions. Here, rows correspond to different solutions and columns correspond to different actions. Each column in the argument to `solution` should be named according to a different action in `x`. Cell values indicate if an action is funded in a given solution or not, and should be either zero or one. Arguments to `solution` can contain additional columns, and they will be ignored.

Value

A `tibble` table containing the following columns:

`"cost"` `numeric` cost of each solution.

`"obj"` `numeric` objective value for each solution. This is calculated using the objective function defined for the argument to `x`.

`x$project_names()` `numeric` column for each project indicating if it was completely funded (with a value of 1) or not (with a value of 0).

`x$feature_names()` `numeric` column for each feature indicating the probability that it will persist into the future given each solution.

See Also

`objectives`, `replacement_costs`, `project_cost_effectiveness`.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# print project data
print(sim_projects)

# print action data
print(sim_features)

# print feature data
print(sim_actions)

# build problem
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 400) %>%
     add_feature_weights("weight") %>%
     add_binary_decisions()
```

```
# print problem
print(p)

# create a table with some solutions
solutions <- data.frame(F1_action =      c(0, 1, 1),
                        F2_action =      c(0, 1, 0),
                        F3_action =      c(0, 1, 1),
                        F4_action =      c(0, 1, 0),
                        F5_action =      c(0, 1, 1),
                        baseline_action = c(1, 1, 1))

# print the solutions
# the first solution only has the baseline action funded
# the second solution has every action funded
# the third solution has only some actions funded
print(solutions)

# calculate statistics
solution_statistics(p, solutions)
```

---

solve *Solve*

Description

Solve a conservation planning `problem`.

Arguments

a	`ProjectProblem-class` or an `OptimizationProblem-class` object.
b	`Solver-class` object. Not used if a is an `ProjectProblem-class` object.
...	arguments passed to `compile`.

Value

The type of object returned from this function depends on the argument to `a`. If the argument to `a` is an `OptimizationProblem-class` object, then the solution is returned as a `list` containing the prioritization and additional information (e.g. run time, solver status). On the other hand, if the argument to `a` is an `ProjectProblem-class` object, then a `tibble` table object will be returned. In this table, each row row corresponds to a different solution and each column describes a different property or result associated with each solution:

`"solution"` `integer` solution identifier.

`"status"` `character` describing each solution. For example, is the solution optimal, suboptimal, or was it returned because the solver ran out of time?

`"obj"` `numeric` objective value for each solution. This is calculated using the objective function defined for the argument to `x`.

`"cost"` `numeric` total cost associated with each solution.

`x$action_names()` `numeric` column for each action indicating if they were funded in each solution or not.

`x$project_names()` `numeric` column for each project indicating if it was completely funded (with a value of 1) or not (with a value of 0).

`x$feature_names()` `numeric` column for each feature indicating the probability that it will persist into the future given each solution.

See Also

problem, solution_statistics, solvers.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# print project data
print(sim_projects)

# print action data
print(sim_features)

# print feature data
print(sim_actions)

# build problem
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 400) %>%
     add_feature_weights("weight") %>%
     add_binary_decisions()

# print problem
print(p)


# solve problem
s <- solve(p)

# print output
print(s)

# print the solver status
print(s$obj)

# print the objective value
print(s$obj)

# print the solution cost
print(s$cost)
```

```
# print which actions are funded in the solution
s[, sim_actions$name, drop = FALSE]

# print the expected probability of persistence for each feature
# if the solution were implemented
s[, sim_features$name, drop = FALSE]
```

Solver-class *Solver prototype*

Description

This prototype is used to generate objects that represent methods for solving optimization problems. **This class represents a recipe to create solver and and is only recommended for use by expert users. To customize the method used to solve optimization problems, please see the help page on** `solvers`.

Fields

$name `character` name of solver.

$parameters `Parameters` object with parameters used to customize the the solver.

$solve `function` used to solve a `OptimizationProblem-class` object.

Usage

`x$print()`

`x$show()`

`x$repr()`

`x$solve(op)`

Arguments

x `Solver-class` object.

op `OptimizationProblem-class` object.

Details

print print the object.

show show the object.

repr `character` representation of object.

solve solve an `OptimizationProblem-class` using this object.

solvers *Problem solvers*

Description

Specify the software and configuration used to solve a project prioritization problem. By default, the best available exact algorithm solver will be used.

Details

The following solvers can be used to find solutions for a project prioritization problem:

add_default_solver This solver uses the best software currently installed on the system.

add_gurobi_solver *Gurobi* is a state-of-the-art commercial optimization software with an R package interface. It is by far the fastest solver that can be used by 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.

add_rsymphony_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.

add_lpsymphony_solver The **lpsymphony** package provides a different interface to the COIN-OR software suite. Unlike the **Rsymhpony** package, the **lpsymphony** package is distributed through Bioconductor. The **lpsymphony** package may be easier to install on Windows or Max OSX systems than the **Rsymphony** package.

add_lpsolveapi_solver *lp_solve* is an open-source integer programming solver. The **lpSolveAPI** package provides an interface to this solver and is available on CRAN. Although this solver is the slowest currently supported solver, it is also the only exact algorithm solver that can be installed on all operating systems without any manual installation steps.

add_heuristic_solver Generate solutions using a backwards heuristic algorithm. Although these types of algorithms have conventionally been used to solve project prioritization problems, they are extremely unlikely to identify optimal solutions and provide no guarantees concerning solution quality.

add_random_solver Generate solutions by randomly funding actions. This can be useful when evaluating the performance of a funding scheme—though it is strongly recommended to evaluate the performance of a funding scheme by comparing it to an optimal solution identified using exact algorithms (e.g. add_gurobi_solver, add_rsymphony_solver).

See Also

constraints, decisions, objectives, problem, targets.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem
p1 <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
     add_binary_decisions()

# build another problem, with the default solver
p2 <- p1 %>%
      add_default_solver()

# build another problem, with the gurobi solver

p3 <- p1 %>%
      add_gurobi_solver()


# build another problem, with the Rsympony solver

p4 <- p1 %>%
      add_rsymphony_solver()


# build another problem, with the lpsymphony solver

p5 <- p1 %>%
      add_lpsymphony_solver()


# build another problem, with the lpSolveAPI solver
p6 <- p1 %>%
      add_lpsolveapi_solver()

# build another problem, with the heuristic solver
p7 <- p1 %>%
      add_heuristic_solver()

# build another problem, with the random solver
p8 <- p1 %>%
      add_random_solver()


# generate solutions using each of the solvers
s <- rbind(solve(p2), solve(p3), solve(p4), solve(p5), solve(p6), solve(p7),
           solve(p8))
s$solver <- c("default", "gurobi", "Rsymphony", "lpsymphony", "lpSolveAPI",
              "heuristic", "random")

# print solutions
```

```
print(as.data.frame(s))
```

`Target-class` *Target prototype*

Description

This prototype is used to represent the targets used when making a prioritization. This prototype inherits from the `ProjectModifier-class`. **This class represents a recipe, to actually add targets to a planning problem, see the help page on `targets`. Only experts should use this class directly.**

See Also

`ProjectModifier-class`.

`targets` *Targets*

Description

Targets are used to specify the minimum probability of persistence required for each feature. Please note that only some objectives require targets, and attempting to solve a problem that requires targets will throw an error if targets are not supplied, and attempting to solve a problem that does not require targets will throw a warning if targets are supplied.

Details

The following functions can be used to specify targets for a project prioritization `problem`:

`add_relative_targets` Set targets as a proportion (between 0 and 1) of the maximum probability of persistence associated with the best project for each feature. For instance, if the best project for a feature has an 80% probability of persisting, setting a 50% (i.e. `0.5`) relative target will correspond to a 40% threshold probability of persisting.

`add_absolute_targets` Set targets by specifying exactly what probability of persistence is required for each feature. For instance, setting an absolute target of 10% (i.e. `0.1`) corresponds to a threshold 10% probability of persisting.

`add_manual_targets` Set targets by manually specifying all the required information for each target.

See Also

`constraints`, `decisions`, `objectives`, `problem`, `solvers`.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with minimum set objective and targets that require each
# feature to have a 30% chance of persisting into the future
p1 <- problem(sim_projects, sim_actions, sim_features,
              "name", "success", "name", "cost", "name") %>%
      add_min_set_objective() %>%
      add_absolute_targets(0.3) %>%
      add_binary_decisions()

# print problem
print(p1)

# build problem with minimum set objective and targets that require each
# feature to have a level of persistence that is greater than or equal to
# 30% of the best project for conserving it
p2 <- problem(sim_projects, sim_actions, sim_features,
              "name", "success", "name", "cost", "name") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.3) %>%
      add_binary_decisions()

# print problem
print(p2)


# solve problems
s1 <- solve(p1)
s2 <- solve(p2)

# print solutions
print(s1)
print(s2)

# plot solutions
plot(p1, s1)
plot(p2, s2)
```

---

tibble-methods *Manipulate tibbles*

Description

Assorted functions for manipulating tibble objects.

Usage

```
## S4 method for signature 'tbl_df'
nrow(x)

## S4 method for signature 'tbl_df'
ncol(x)

## S4 method for signature 'tbl_df'
as.list(x)
```

Arguments

`x`	`tibble` object.

Details

The following methods are provided from manipulating `tibble` objects.

nrow extract `integer` number of rows.

ncol extract `integer` number of columns.

as.list convert to a `list`.

print print the object.

Examples

```
# load tibble package
require(tibble)

# make tibble
a <- tibble(value = seq_len(5))

# number of rows
nrow(a)

# number of columns
ncol(a)

# convert to list
as.list(a)
```

`Weight-class` *Weight prototype*

Description

This prototype is used to represent the weights used when making a prioritization. This prototype inherits from the `ProjectModifier-class`. **This class represents a recipe, to actually add targets to a planning problem, see the help page on `weights`. Only experts should use this class directly.**

See Also

ProjectModifier-class.

---

weights *Weights*

Description

Weights are used to specify the relative importance for specific features persisting into the future. Please note that only some objectives require weights, and attempting to solve a problem that does not require weights will throw a warning and the weights will be ignored.

Details

Currently, only one function can be used to specify weights:

add_feature_weights Set feature weights for a project prioritization problem.

See Also

constraints, decisions, objectives, problem, solvers, targets.

Examples

```
# load data
data(sim_projects, sim_features, sim_actions)

# build problem with maximum richness objective, $300 budget, and
# feature weights
p <- problem(sim_projects, sim_actions, sim_features,
             "name", "success", "name", "cost", "name") %>%
     add_max_richness_objective(budget = 200) %>%
     add_feature_weights("weight") %>%
     add_binary_decisions()


# solve problem
s <- solve(p)

# print solution
print(s)

# plot solution
plot(p, s)
```

%>% — *Pipe operator*

Description

This package uses the pipe operator (`%>%`) to express nested code as a series of imperative procedures.

Arguments

`lhs, rhs`	An object and a function.

See Also

`%>%`, `tee`.

Examples

```
# set seed for reproducibility
set.seed(500)

# generate 100 random numbers and calculate the mean
mean(runif(100))

# reset the seed
set.seed(500)

# repeat the previous procedure but use the pipe operator instead of nesting
# function calls inside each other.
runif(100) %>% mean()
```

%T>% — *Tee operator*

Description

This package uses the "tee" operator (`%T>%`) to modify objects.

Arguments

`lhs, rhs`	An object and a function.

See Also

`%T>%`, `pipe`.

Examples

```
# the tee operator returns the left-hand side of the result and can be
# useful when dealing with mutable objects. In this example we want
# to use the function "f" to modify the object "e" and capture the
# result

# create an empty environment
e <- new.env()

# create a function to modify an environment and return NULL
f <- function(x) {x$a <- 5; return(NULL)}

# if we use the pipe operator we won't capture the result since "f"()
# returns a NULL
e2 <- e %>% f()
print(e2)

# but if we use the tee operator then the result contains a copy of "e"
e3 <- e %T>% f()
print(e3)
```

Index