Package 'oppr'

May 27, 2026

Type Package

Version 1.1.0

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. The 'rcbc' R package (available at <https://github.com/dirkschumacher/rcbc>) can also be used to generate solutions using the CBC optimization software (<https://github.com/coin-or/Cbc>). Finally, methods are provided for comparing different prioritizations and evaluating their benefits. For more information, see Hanson et al. (2019) <doi:10.1111/2041-210X.13264>.

Imports utils, methods, stats, Matrix (>= 1.3-0), magrittr (>= 1.5), R6 (>= 2.5.1), cli (>= 1.0.1), assertthat (>= 0.2.0), tibble (>= 2.0.0), ape (>= 5.2), tidytree (>= 0.3.3), ggplot2 (>= 3.5.0), highs (>= 1.10.0.3), viridisLite (>= 0.3.0), withr (>= 2.4.1), rlang (>= 1.1.3)

Suggests testthat (>= 2.0.0), knitr (>= 1.20), roxygen2 (>= 6.1.0), rmarkdown (>= 1.10), gurobi (>= 13.0.0), Rsymphony (>= 0.1.28), ggtree (>= 2.4.2), lpsymphony (>= 1.10.0), lpSolveAPI (>= 5.5.2.0.17), rcbc (>= 0.1.0.9003), fansi (>= 1.0.6)

Depends R(>= 3.5.0)

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

License GPL-3

LazyData true

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

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

VignetteBuilder knitr, rmarkdown

Encoding UTF-8

Language en-US

Collate 'internal.R' 'waiver.R' 'ProjectProblem-class.R'
'ProjectModifier-class.R' 'Constraint-class.R'
'Decision-class.R' 'MultiObjApproach-class.R'
'MultiObjProjectProblem-class.R' 'Objective-class.R'
'OptimizationProblem-class.R' 'OptimizationProblem-methods.R'
'RcppExports.R' 'Solver-class.R' 'Target-class.R'
'Weight-class.R' 'action_names.R'
'add_abs_constraint_approach.R' 'add_absolute_targets.R'
'add_binary_decisions.R' 'add_cbc_solver.R'
'add_default_solver.R' 'add_default_weights.R'
'add_feature_weights.R' 'add_gurobi_solver.R'
'add_heuristic_solver.R' 'add_highs_solver.R'
'add_locked_in_action_constraints.R'
'add_locked_in_project_constraints.R'
'add_locked_out_action_constraints.R'
'add_locked_out_project_constraints.R'
'add_lpsolveapi_solver.R' 'add_lpsymphony_solver.R'
'add_manual_locked_action_constraints.R'
'add_manual_locked_project_constraints.R' 'tbl_df.R'
'add_manual_targets.R' 'add_max_phylo_div_objective.R'
'star_phylogeny.R' 'add_max_richness_objective.R'
'add_max_targets_met_objective.R' 'add_max_wtd_sum_objective.R'
'add_min_set_objective.R' 'add_random_solver.R'
'add_ref_point_approach.R' 'add_relative_targets.R'
'add_rsymphony_solver.R' 'add_wtd_goal_approach.R'
'approaches.R' 'assertions.R' 'branch_matrix.R' 'compile.R'
'constraints.R' 'decisions.R' 'deprecated.R' 'feature_names.R'
'multi_compile.R' 'multi_problem.R'
'new_optimization_problem.R' 'number_of_actions.R'
'number_of_features.R' 'number_of_problems.R'
'number_of_projects.R' 'objectives.R' 'package.R'
'solution_statistics.R' 'plot.R' 'plot_solution_barplot.R'
'plot_solution_phylogram.R' 'predefined_optimization_problem.R'
'print.R' 'problem.R' 'problem_names.R'
'project_cost_effectiveness.R' 'project_names.R'
'rake_phylogeny.R' 'rank_importance.R' 'reexports.R'
'replacement_costs.R' 'repr.R' 'run_example.R' 'show.R'
'sim_data.R' 'sim_multi_data.R' 'simulate_multi_ppp_data.R'
'simulate_ppp_data.R' 'simulate_ptm_data.R' 'solve.R'
'solvers.R' 'targets.R' 'weights.R' 'zzz.R'

Config/testthat/edition 3

Config/Needs/website tidyr (>= 0.8.2)

Config/roxygen2/version 8.0.0

NeedsCompilation yes

Author Jeffrey O Hanson [aut, cre] (ORCID:
<https://orcid.org/0000-0002-4716-6134>),
Richard Schuster [aut] (ORCID: <https://orcid.org/0000-0003-3191-7869>),
Matthew Strimas-Mackey [aut] (ORCID:
<https://orcid.org/0000-0001-8929-7776>),
Joseph R Bennett [aut] (ORCID: <https://orcid.org/0000-0002-3901-9513>)

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

Repository CRAN

Date/Publication 2026-05-27 07:00:15 UTC

Contents

`action_names` *Action names*

Description

Get the names of actions in an object.

Usage

```
action_names(x)

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

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

Arguments

x	`problem()` or `multi_problem()` object.

Value

A `character` vector.

Examples

```
# 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_wtd_sum_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

Add targets to a project prioritization that specify the desired expected outcome for each feature in the same units as the outcomes values. For example, if a feature has its outcome values expressed probabilities of persistence, then setting an absolute target of 0.1 means that the feature should ideally have a 10% chance of persistence.

Usage

```
add_absolute_targets(x, targets)

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

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

Arguments

`x`	[problem() object.
`targets`	Object that specifies the targets for each feature. See the Details section for more information.

Details

Targets are used to specify a threshold minimum desirable expected outcome for each feature. These should ideally be set according to stakeholder requirements and expert knowledge. 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 using the following options.

`numeric` **value** The value is used to set the target threshold for each feature. This option may be useful when all features should be assigned the same target threshold.

`numeric` **vector** Each value specifies a target threshold for each feature. The order of the values should correspond to the order of the features in `x`.

`character` **value** The value specifies the name of a column in the feature data (i.e., the argument to `features` in the `problem()` function). The target threshold for each feature is set according the column values.

Value

A `problem()` object with the targets added to it.

See Also

Other targets: add_manual_targets(), add_relative_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_abs_constraint_approach` *Add an absolute constraint approach*

Description

Add an constraint approach for multi-objective optimization to a project problem based on the required objective values.

Usage

```
add_abs_constraint_approach(x, goals, verbose = TRUE)
```

Arguments

`x`	`multi_problem()` object.
`goals`	`numeric` vector containing values that denote the represent a threshold minimum level of achievement for each objective. If no goal is required for a particular objective, then a missing `NA` value can be specified. To generate multiple solutions based on different values, `goals` can be a `numeric` matrix where each row corresponds to a different solution and each columns corresponds to a different objective.
`verbose`	`logical` should progress on generating solutions displayed? Defaults to `TRUE`.

Details

Constraint-based approaches for multi-objective optimization involves adding constraints to a problem formulation to ensure that solutions achieve a particular level of performance each objective, whilst maximizing performance according to a primary objective. In particular, each objective is

associated with a goal that specifies a threshold minimum level of performance (conceptually similar to a target in the minimum set formulation). Below we provide the mathematical details for this approach.

To describe this approach mathematically, we will define the following terminology. Let O denote the set of objectives (indexed by o). For each objective, G_o denote the goal for each objective $o \in O$, and V_o denote the objective value for a candidate solution as measured based on each objective $o \in O$. Also, let V_1 denote the objective value for the first objective. Although we assume that all objectives here should be maximized (for brevity), this approach is compatible with objectives that should also be minimized. After defining these terms, the approach is formulated with the following equation.

$$\text{Maximize} V_1 \text{Subject to} V_o \geq G_o \forall o \in O$$

Value

A `multi_problem()` object with the approach added to it.

See Also

Other approaches: `add_ref_point_approach()`, `add_wtd_goal_approach()`

Examples

```
# load data
data(sim_multi_projects)
data(sim_multi_features)
data(sim_multi_actions)
data(sim_multi_tree)

# build problem
p <-
  multi_problem(
    obj1 =
      problem(
        sim_multi_projects[[1]], sim_multi_actions, sim_multi_features[[1]],
        "name", "success", "name", "cost", "name",
        baseline_project_name = "baseline_project_obj1"
      ) %>%
      add_max_phylo_div_objective(
        budget = 1000, tree = sim_multi_tree[[1]]
      ) %>%
      add_binary_decisions(),
    obj2 =
      problem(
        sim_multi_projects[[2]], sim_multi_actions, sim_multi_features[[2]],
        "name", "success", "name", "cost", "name",
        baseline_project_name = "baseline_project_obj2"
      ) %>%
      add_max_richness_objective(budget = 1000) %>%
      add_binary_decisions(),
    obj3 =
```

```
    problem(
      sim_multi_projects[[3]], sim_multi_actions, sim_multi_features[[3]],
      "name", "success", "name", "cost", "name",
      baseline_project_name = "baseline_project_obj3"
    ) %>%
    add_max_wtd_sum_objective(budget = 1000) %>%
    add_binary_decisions()
) %>%
add_abs_constraint_approach(goals = c(NA, 0.01, 0.01)) %>%
add_default_solver()

# print problem
print(p)

# solve problem
s <- solve(p)

# print solution
print(s)
```

`add_binary_decisions` *Add binary decisions*

Description

Add binary decisions to a project prioritization problem. This means that the optimization process aims to determine if each action should be selected for funding or not.

Usage

```
add_binary_decisions(x)
```

Arguments

x	`problem()` object.

Details

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

A `problem()` object with the decisions added to it.

Examples

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

# build problem with maximum weighted sum objective, $200 budget, and
# binary decisions
p <-
  problem(
    sim_projects, sim_actions, sim_features,
    "name", "success", "name", "cost", "name"
  ) %>%
  add_max_wtd_sum_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_cbc_solver *Add a* CBC *solver*

Description

Add a solver to generate solutions to a project prioritization problem with the *CBC* (COIN-OR branch and cut) Forrest & Lougee-Heimer 2005). This function can also be used to customize the behavior of the solver. It requires the **rcbc** package to be installed (only available on GitHub, see below for installation instructions).

Usage

```
add_cbc_solver(
  x,
  gap = 0.1,
  time_limit = .Machine$integer.max,
  presolve = 2,
  threads = 1,
  first_feasible = FALSE,
  start = NULL,
  verbose = TRUE
)
```

Arguments

x	problem() or multi_problem() object.
gap	numeric gap to optimality. This gap is relative and expresses the acceptable deviance from the optimal objective. For example, a value of 0.01 will result in the solver stopping when it has found a solution within 1% of optimality. Additionally, a value of 0 will result in the solver stopping when it has found an optimal solution. The default value is 0 (i.e., 0% from optimality).
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. Available options are: (0) disable pre-solving, (1) conservative level of pre-solving, and (2) very aggressive level of pre-solving . The default value is 2.
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.
start	logical vector with (TRUE/FALSE) values for each action indicating if they should be selected by the starting solution. These values should be in the same order of the actions in x (i.e., per action_names(x)). Missing (NA) values can be used to indicate that the solver should automatically calculate starting values for particular actions. Defaults to NULL such that starting values are automatically determined by the solver for all actions.
verbose	logical should information be printed during optimization? Defaults to TRUE.

Details

CBC is an open-source mixed integer programming solver that is part of the Computational Infrastructure for Operations Research (COIN-OR) project. This solver seems to have much better performance than the other open-source solvers (i.e., add_highs_solver(), add_rsymphony_solver(), add_lpsymphony_solver()) (see the *Solver benchmarks* vignette for details). As such, it is strongly recommended to use this solver if the *Gurobi* solver is not available.

Value

A problem() object with the solver added to it.

Installation

The **rcbc** package is required to use this solver. Since the **rcbc** package is not available on the the Comprehensive R Archive Network (CRAN), it must be installed from its GitHub repository. To install the **rcbc** package, please use the following code:

```
if (!require(remotes)) install.packages("remotes")
remotes::install_github("dirkschumacher/rcbc")
```

Note that you may also need to install several dependencies – such as the Rtools software or system libraries – prior to installing the **rcbc** package. For further details on installing this package, please consult the online package documentation.

References

Forrest J and Lougee-Heimer R (2005) CBC User Guide. In Emerging theory, Methods, and Applications (pp. 257–277). INFORMS, Catonsville, MD. doi:10.1287/educ.1053.0020.

See Also

Other solvers: `add_default_solver()`, `add_gurobi_solver()`, `add_heuristic_solver()`, `add_highs_solver()`, `add_lpsolveapi_solver()`, `add_lpsymphony_solver()`, `add_random_solver()`, `add_rsymphony_solver()`

Examples

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

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

# 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

Add the best solver currently installed to a project prioritization problem.

Usage

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

Arguments

x	problem() or multi_problem() object.
...	arguments passed to the solver.

Details

The solvers that can be used are as follows (ordered best to worst): **gurobi**, (add_gurobi_solver()), **highs**, (add_highs_solver()), **rcbc**, (add_cbc_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.

Value

A problem() object with the solver added to it.

See Also

Other solvers: add_cbc_solver(), add_gurobi_solver(), add_heuristic_solver(), add_highs_solver(), add_lpsolveapi_solver(), add_lpsymphony_solver(), add_random_solver(), add_rsymphony_solver()

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_wtd_sum_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_default_weights` *Add default weights*

Description

Add the default weights to a project prioritization problem.

Usage

```
add_default_weights(x)
```

Arguments

x	`problem()` object.

Value

A `problem()` with the weights added to it.

See Also

Other weights: `add_feature_weights()`

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_wtd_sum_objective(budget = 200) %>%
  add_default_weights() %>%
```

```
  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

Add weights to the features in a project prioritization problem.

Usage

```
add_feature_weights(x, weights)

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

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

Arguments

`x`	`problem()` 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 of particular features. For budget constrained problems (e.g., `add_max_wtd_sum_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 weight values range between 0.001 and 1,000 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** Values specify a weight for each feature. These `numeric` values should correspond to each row in the argument to `features` for `problem()`.

`character` **value** The value specifies the name of a column in the feature data (i.e., argument to `features` in `problem()`). The column must have `numeric` values, and these used to weight the features.

Value

A `problem()` with the weights added to it.

See Also

See weights for an overview of functions for adding weights.

Other weights: `add_default_weights()`

Examples

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

# print feature data
print(sim_features)

# build problem with maximum weighted sum objective, $300 budget,
# and no weights
p1 <-
  problem(
    sim_projects, sim_actions, sim_features,
    "name", "success", "name", "cost", "name"
  ) %>%
  add_max_wtd_sum_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

Add a solver to generate solutions to a project prioritization problem with the *Gurobi* software. This function can also be used to customize the behavior of the solver. See below for details on installation requirements.

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 = FALSE,
  numeric_focus = 1,
  start = NULL,
  verbose = TRUE
)
```

Arguments

x	problem() or multi_problem() object.
gap	numeric gap to optimality. This gap is relative and expresses the acceptable deviance from the optimal objective. For example, a value of 0.01 will result in the solver stopping when it has found a solution within 1% of optimality. Additionally, a value of 0 will result in the solver stopping when it has found an optimal solution. The default value is 0 (i.e., 0% from optimality).
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., https://docs.gurobi.com/projects/optimizer/en/current/reference/parameters.html#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.
numeric_focus	integer value denoting how much extra attention be paid to verifying the accuracy of numerical calculations? Acceptable values include 0, 1, 2, or 3. This may be useful when dealing with problems that may suffer from numerical instability issues. Beware that setting greater values will likely increase run time. Defaults to 1.
start	logical vector with (TRUE/FALSE) values for each action indicating if they should be selected by the starting solution. These values should be in the same order of the actions in x (i.e., per action_names(x)). Missing (NA) values can be used to indicate that the solver should automatically calculate starting values

for particular actions. Defaults to NULL such that starting values are automatically determined by the solver for all actions.

verbose `logical` should information be printed during optimization? Defaults to TRUE.

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 https://support.gurobi.com/hc/en-us/articles/4534161999889-How-do-I-install-Gurobi-Optimizer for instructions). 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 https://support.gurobi.com/hc/en-us/articles/14462206790033-How-do-I-install-Gurobi-for-R for instructions).

Value

A `problem()` object with the solver added to it.

See Also

See solvers for an overview of functions for adding solvers.

Other solvers: `add_cbc_solver()`, `add_default_solver()`, `add_heuristic_solver()`, `add_highs_solver()`, `add_lpsolveapi_solver()`, `add_lpsymphony_solver()`, `add_random_solver()`, `add_rsymphony_solver()`

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_wtd_sum_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

Add a solver to a project prioritization problem to generate solutions 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_highs_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`	`problem()` or `multi_problem()` 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. Although this step can improve solution quality, previous algorithms for project prioritization have not used it (e.g., Probert *et al.* 2016). Defaults to `TRUE`.

`verbose` `logical` should information be printed during optimization? Defaults to `TRUE`.

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_wtd_sum_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:

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

Here 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:

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

Here 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

A `problem()` 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). Formerly available at https://github.com/p-robot (copy available at https://github.com/jeffreyhanson/ppp).

See Also

Other solvers: `add_cbc_solver()`, `add_default_solver()`, `add_gurobi_solver()`, `add_highs_solver()`, `add_lpsolveapi_solver()`, `add_lpsymphony_solver()`, `add_random_solver()`, `add_rsymphony_solver()`

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_wtd_sum_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_highs_solver` *Add a* HiGHS *solver*

Description

Add a solver to generate solutions to a project prioritization problem with the *HiGHS* software (Huangfu and Hall 2018). This function can also be used to customize the behavior of the solver. It requires the **highs** package to be installed.

Usage

```
add_highs_solver(
  x,
  gap = 0,
  time_limit = .Machine$integer.max,
  presolve = TRUE,
  threads = 1,
  start = NULL,
  verbose = TRUE,
```

```
  control = list()
)
```

Arguments

x	problem() or multi_problem() object.
gap	numeric gap to optimality. This gap is relative and expresses the acceptable deviance from the optimal objective. For example, a value of 0.01 will result in the solver stopping when it has found a solution within 1% of optimality. Additionally, a value of 0 will result in the solver stopping when it has found an optimal solution. The default value is 0 (i.e., 0% from optimality).
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.
start	logical vector with (TRUE/FALSE) values for each action indicating if they should be selected by the starting solution. These values should be in the same order of the actions in x (i.e., per action_names(x)). Missing (NA) values can be used to indicate that the solver should automatically calculate starting values for particular actions. Defaults to NULL such that starting values are automatically determined by the solver for all actions.
verbose	logical should information be printed during optimization? Defaults to TRUE.
control	list with additional parameters for tuning the optimization process. For example, control = list(simplex_strategy = 1) could be used to set the simplex_strategy parameter. See the online documentation for information on the parameters.

Value

A problem() object with the solver added to it.

References

Huangfu Q and Hall JAJ (2018). Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10: 119-142.

See Also

Other solvers: add_cbc_solver(), add_default_solver(), add_gurobi_solver(), add_heuristic_solver(), add_lpsolveapi_solver(), add_lpsymphony_solver(), add_random_solver(), add_rsymphony_solver()

Examples

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

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

# print problem
print(p)

# solve problem
s <- solve(p)

# print solution
print(s)

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

`add_locked_in_action_constraints` *Add locked in action constraints*

Description

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

Usage

```
add_locked_in_action_constraints(x, locked_in)

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

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

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

Arguments

x	problem() 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** Each values specifies the index for an action that should be locked when generating solutions (i.e., row numbers of the actions in the argument to actions in problem()).

logical **vector** Each value (i.e., TRUE and/or FALSE) indicates if an action should be locked (or not) when generating the solution. These logical values should correspond to each row in the argument to actions in problem().

character **value** The value specifies the name of a column in the action data (i.e., argument to actions in problem()). The column must have logical (i.e., TRUE and/or FALSE) values, and these values are used to indicate which actions are locked (or not).

Value

A problem() object with the constraints added to it.

See Also

See constraints for an overview of functions for adding constraints.

Other constraints: add_locked_in_project_constraints(), add_locked_out_action_constraints(), add_locked_out_project_constraints(), add_manual_locked_action_constraints(), add_manual_locked_proje

Examples

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

# print action data
print(sim_actions)

# build problem with maximum weighted sum objective and $150 budget
p1 <-
  problem(
    sim_projects, sim_actions, sim_features,
    "name", "success", "name", "cost", "name"
  ) %>%
  add_max_wtd_sum_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_action_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_action_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
p4 <- p1 %>% add_locked_in_action_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_in_project_constraints` *Add locked in project constraints*

Description

Add constraints to a project prioritization problem to ensure that particular projects are selected for funding by the solution. For example, it may be desirable to lock in projects for conserving culturally or taxonomically important species.

Usage

```
add_locked_in_project_constraints(x, locked_in)

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

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

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

Arguments

x	problem() object.
locked_in	Object that determines which planning units that should be locked in. See the Details section for more information.

Details

The locked projects can be specified in several different ways:

`integer` **vector** Each values specifies the index for a project that should be locked when generating solutions (i.e., row numbers of the projects in the argument to `projects` in problem()).

`logical` **vector** Each value (i.e., `TRUE` and/or `FALSE`) indicates if a project should be locked (or not) when generating the solution. These `logical` values should correspond to each row in the argument to `projects` in problem().

`character` **value** The value specifies the name of a column in the project data (i.e., argument to `projects` in problem()). The column must have `logical` (i.e., `TRUE` and/or `FALSE`) values, and these values are used to indicate which projects are locked (or not).

Value

A problem() object with the constraints added to it.

See Also

Other constraints: add_locked_in_action_constraints(), add_locked_out_action_constraints(), add_locked_out_project_constraints(), add_manual_locked_action_constraints(), add_manual_locked_proje

Examples

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

# add column to the project data to indicate which projects should be
# locked. in particular, this column will lock the first project
sim_projects$locked_in <- c(TRUE, rep(FALSE, nrow(sim_projects) - 1))

# print project data
print(sim_projects)

# build problem with maximum weighted sum objective and $150 budget
p1 <-
  problem(
    sim_projects, sim_actions, sim_features,
```

```
    "name", "success", "name", "cost", "name"
  ) %>%
  add_max_wtd_sum_objective(budget = 150) %>%
  add_binary_decisions()

# print problem
print(p1)

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

# print problem
print(p2)

# build another problem, and lock in the projects using logical inputs from
# the sim_projects stable
p3 <- p1 %>% add_locked_in_project_constraints(sim_projects$locked_in)

# print problem
print(p3)

# build another problem, and lock in the projects using the column name
# "locked_in" in the sim_projects table
p4 <- p1 %>% add_locked_in_project_constraints("locked_in")

# print problem
print(p4)

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

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

`add_locked_out_action_constraints` *Add locked out action constraints*

Description

Add constraints to a project prioritization problem to ensure that particular actions are not selected for funding by 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_action_constraints(x, locked_out)

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

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

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

Arguments

x	problem() object.
locked_out	Object that determines which planning units that should be locked out. See the Details section for more information.

See Also

Other constraints: add_locked_in_action_constraints(), add_locked_in_project_constraints(), add_locked_out_project_constraints(), add_manual_locked_action_constraints(), add_manual_locked_proje

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 weighted sum objective and $150 budget
p1 <-
  problem(
    sim_projects, sim_actions, sim_features,
    "name", "success", "name", "cost", "name"
  ) %>%
  add_max_wtd_sum_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_action_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_action_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
p4 <- p1 %>% add_locked_out_action_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_locked_out_project_constraints` *Add locked out project constraints*

Description

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

Usage

```
add_locked_out_project_constraints(x, locked_out)

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

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

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

Arguments

x	problem() object.
locked_out	Object that determines which planning units that should be locked out. See the Details section for more information.

See Also

Other constraints: add_locked_in_action_constraints(), add_locked_in_project_constraints(), add_locked_out_action_constraints(), add_manual_locked_action_constraints(), add_manual_locked_projec

Examples

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

# add column to the project data to indicate which projects should be
# locked. in particular, this column will lock the first project
sim_projects$locked_out <- c(TRUE, rep(FALSE, nrow(sim_projects) - 1))

# print project data
print(sim_projects)

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

# print problem
print(p1)

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

# print problem
print(p2)

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

# print problem
print(p3)

# build another problem, and lock out the projects using the column name
```

```
# "locked_out" in the sim_projects table
p4 <- p1 %>% add_locked_out_project_constraints("locked_out")

# print problem
print(p4)

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

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

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

Description

Add a solver to generate solutions to a project prioritization problem with the *lp_solve* software. This function can also be used to customize the behavior of the solver. It requires the **lpSolveAPI** package to be installed.

Usage

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

Arguments

x	problem() or multi_problem() object.
gap	numeric gap to optimality. This gap is relative and expresses the acceptable deviance from the optimal objective. For example, a value of 0.01 will result in the solver stopping when it has found a solution within 1% of optimality. Additionally, a value of 0 will result in the solver stopping when it has found an optimal solution. The default value is 0 (i.e., 0% from optimality).
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 during optimization? Defaults to TRUE.

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

A `problem()` object with the solver added to it.

See Also

Other solvers: `add_cbc_solver()`, `add_default_solver()`, `add_gurobi_solver()`, `add_heuristic_solver()`, `add_highs_solver()`, `add_lpsymphony_solver()`, `add_random_solver()`, `add_rsymphony_solver()`

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_wtd_sum_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_lpsymphony_solver` *Add a* SYMPHONY *solver with* lpsymphony

Description

Add a solver to generate solutions to a project prioritization problem with the *SYMPHONY* software. This function can also be used to customize the behavior of the solver. It requires the **lpsymphony** package to be installed.

Usage

```
add_lpsymphony_solver(
  x,
  gap = 0,
  time_limit = .Machine$integer.max,
  first_feasible = FALSE,
  verbose = TRUE
)
```

Arguments

x	problem() or multi_problem() object.
gap	numeric gap to optimality. This gap is relative and expresses the acceptable deviance from the optimal objective. For example, a value of 0.01 will result in the solver stopping when it has found a solution within 1% of optimality. Additionally, a value of 0 will result in the solver stopping when it has found an optimal solution. The default value is 0 (i.e., 0% from optimality).
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 during optimization? Defaults to TRUE.

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

A problem() object with the solver added to it.

See Also

Other solvers: add_cbc_solver(), add_default_solver(), add_gurobi_solver(), add_heuristic_solver(), add_highs_solver(), add_lpsolveapi_solver(), add_random_solver(), add_rsymphony_solver()

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_wtd_sum_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_action_constraints

Add manually specified locked constraints for actions

Description

Add constraints to a project prioritization problem to ensure that particular actions are selected, or not selected, for funding by the solution. This function offers more fine-grained control than the add_locked_in_action_constraints() and add_locked_out_action_constraints() functions.

Usage

```
add_manual_locked_action_constraints(x, locked)

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

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

Arguments

x	problem() object.
locked	data.frame or tibble::tibble() object. See the Details section for more information.

Details

The argument to locked must contain the following columns.

"action" character values with action names.

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

Value

A problem() object with the constraints added to it.

See Also

Other constraints: add_locked_in_action_constraints(), add_locked_in_project_constraints(), add_locked_out_action_constraints(), add_locked_out_project_constraints(), add_manual_locked_project_

Examples

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

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

# print locked statuses
print(locked_data)

# 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_wtd_sum_objective(budget = 500) %>%
  add_manual_locked_action_constraints(locked_data) %>%
  add_binary_decisions()

# print problem
print(p)

# solve problem
```

```
s <- solve(p)

# print solution
print(s)
```

`add_manual_locked_project_constraints` *Add manually specified locked constraints for projects*

Description

Add constraints to a project prioritization problem to ensure that particular projects are selected, or not selected, for funding by the solution. This function offers more fine-grained control than the `add_locked_in_project_constraints()` and `add_locked_out_project_constraints()` functions.

Usage

```
add_manual_locked_project_constraints(x, locked)

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

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

Arguments

`x`	`problem()` object.
`locked`	`data.frame` or `tibble::tibble()` object. See the Details section for more information.

Details

The argument to `locked` must contain the following columns.

`"project"` `character` values with project names.

`"status"` `numeric` values indicating if projects should be selected for funding (with a value of 1) or not (with a value of zero).

Value

A `problem()` object with the constraints added to it.

See Also

Other constraints: `add_locked_in_action_constraints()`, `add_locked_in_project_constraints()`, `add_locked_out_action_constraints()`, `add_locked_out_project_constraints()`, `add_manual_locked_action_c`

Examples

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

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

# print locked statuses
print(locked_data)

# 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_wtd_sum_objective(budget = 500) %>%
  add_manual_locked_project_constraints(locked_data) %>%
  add_binary_decisions()

# print problem
print(p)

# solve problem
s <- solve(p)

# print solution
print(s)
```

`add_manual_targets` *Add manual targets*

Description

Add targets to a project prioritization problem by manually specifying detailed information for each target threshold. Although this function is useful because it can be used to customize all aspects of a target, it requires considerable more information that other functions for adding targets (e.g., `add_absolute_targets()` and `add_relative_targets()`).

Usage

```
add_manual_targets(x, targets)

## 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	[problem() object.
targets	data.frame or tibble::tibble() object. See the Details section for more information.

Details

Targets are used to specify a threshold minimum desirable expected outcome for each feature. These should ideally be set according to stakeholder requirements and expert knowledge. Please note that attempting to solve problems with objectives that require targets without specifying targets will throw an error.

The argument to targets should contain the following columns:

"feature" character values with names of features in x.

"type" character values describing the type of target. Acceptable values include "absolute" and "relative".

"sense" character values indicating the constraint sense for 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

A problem() object with the targets added to it.

See Also

See targets for an overview for functions for adding targets.

Other targets: add_absolute_targets(), add_relative_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

Add an objective to a project prioritization problem based on maximizing phylogenetic diversity, whilst ensuring that the cost of the solution is within a pre-specified budget (Bennett *et al.* 2014, Faith 2008). Note that this objective requires that the outcome data in the `problem()` reflect probabilities of persistence.

Usage

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

Arguments

`x`	`problem()` object.
`budget`	`numeric` value representing the maximum amount of total expenditure for funding actions.
`tree`	`ape::phylo()` phylogenetic tree describing the evolutionary relationships between the features. Note that `tree` must contain every feature, and only the features, present in `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 $Q_{fj} = (P_j \times B_{fj}) + \Big(\big(1 - (P_j B_{fj})\big) \times (P_n \times B_{fn}) \Big)$, 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}, 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 continuous E_f variables denote the probability that feature f will go extinct; and the 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).

$$\text{Maximize} (\sum_{b=0}^{B} L_b R_b) + \sum_{f}^{F} (1 - E_f) W_f \quad \text{(eqn 1a)}$$

$$\text{Subject to} \sum_{i=0}^{I} C_i \leq m \quad \text{(eqn 1b)}$$

$$R_b = 1 - \prod_{f=0}^{F} ifelse(T_{bf} == 1, E_f, 1) \forall b \in$$

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

A `problem()` 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

Other objectives: `add_max_richness_objective()`, `add_max_targets_met_objective()`, `add_max_wtd_sum_objectiv`
`add_min_set_objective()`

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 $200 budget
p1 <-
  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
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

Add an objective to a project prioritization problem based on maximizing the number of features likely to persist, whilst ensuring that the cost of the solution is within a pre-specified budget (Joseph, Maloney & Possingham 2009). Weights can be used to specify the relative importance of conserving specific features (see `add_feature_weights()`). Although this objective is conceptually similar to maximizing the weighted sum of expected outcomes for the features (`add_max_richness_objective()` it is designed to work for features that are associated with probabilistic outcomes.

Usage

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

Arguments

x	`problem()` object.

budget numeric value representing the maximum amount of total expenditure 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 are 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 continuous E_f variables denote the probability that feature f will persist.

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).

$$\text{Maximize} \sum_{f=0}^{F} E_f W_f \text{(eqn1a)} \text{Subject to} \sum_{i=0}^{I} C_i \leq m \text{(eqn1b)} E_f = \sum_{j=0}^{J} Z_{fj} Q_{fj} \forall f \in F \text{(eqn1c)} Z_{fj} \leq Y_j \forall j \in J \text{(eqn1d)} \sum_{j=}$$

The objective (eqn 1a) is to maximize the weighted number of features that are expected to persist. 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

A problem() 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

Other objectives: add_max_phylo_div_objective(), add_max_targets_met_objective(), add_max_wtd_sum_objecti
add_min_set_objective()

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

Add an objective to a project prioritization problem based on maximizing the number of targets met, 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 threshold level of expected outcome for each feature (e.g., a 90% chance of persistence for each feature). In such exercises, the decision makers do not perceive any benefit when a target is not met (e.g., if a feature has a target corresponding to a 90% chance of persistence, then no benefit is accrued if the feature has a 50% chance of persistence), and do not assign any greater benefit for surpassing a target (e.g., if a feature has a target corresponding to a 90% chance of persistence, then the same level of benefit would be accrued if the feature had a 90% or 95% chance of persistence). Furthermore, weights can also be used to specify the relative importance of meeting targets for particular features (see `add_feature_weights()`).

Usage

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

Arguments

x	`problem()` object.
budget	`numeric` value representing the maximum amount of total expenditure 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 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 target for each feature f, and E_f denote the expected outcome for each feature 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 outcome for each feature $f \in F$ associated with the project $j \in J$ assuming that all of the actions comprising project j are funded and that project is allocated to feature f. For convenience, let Q_{fj} denote the expected outcome for each $f \in F$ associated with the project $j \in J$ 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 expected outcome for a feature given that a project is funded and allocated to the feature depends on (i) the probability of the project succeeding, (ii) the outcome for the feature if the project succeeds, and (iii) the outcome for the feature if the project fails (per the baseline project). 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 continuous E_f variables denote the expected outcome for feature f; and the binary G_f variables indicate if the 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).

$$\text{Maximize} \sum_{f=0}^{F} G_f W_f (\text{eqn1a}) \text{Subject to} \sum_{i=0}^{I} C_i \leq m (\text{eqn1b}) E_f \geq G_f T_f \forall f \in F (\text{eqn1c}) E_f = \sum_{j=0}^{J} Z_{fj} Q_{fj} \forall f \in F (\text{eqn1d}$$

The objective (eqn 1a) is to maximize the weighted total number of the features that have their targets met. Constraints (eqn 1b) calculate which 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 expected outcome for each feature 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 expected outcome variables (E_f) are greater than zero. 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

A `problem()` 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

Other objectives: `add_max_phylo_div_objective()`, `add_max_richness_objective()`, `add_max_wtd_sum_objective()`, `add_min_set_objective()`

Examples

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

# load data
data(sim_projects, sim_features, sim_actions)

# manually adjust feature weights
sim_features$weight <- c(8, 2, 6, 3, 1)

# build problem with maximum targets met objective, a $200 budget,
# targets that require each feature to have a 20% chance of persisting into
# the future, and zero cost actions locked in
p1 <-
  problem(
    sim_projects, sim_actions, sim_features,
    "name", "success", "name", "cost", "name"
  ) %>%
  add_max_targets_met_objective(budget = 200) %>%
  add_absolute_targets(0.2) %>%
  add_locked_in_action_constraints(which(sim_actions$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 F5 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 the feature F3 is now funded
# to enhance its probability of persistence
plot(p2, s2) + geom_hline(yintercept = 0.2, linetype = "dashed")
```

add_max_wtd_sum_objective *Add maximum weighted sum objective*

Description

Add an objective to a project prioritization problem based on maximizing the weighted sum of expected outcomes associated with the features, whilst ensuring that the cost of the solution is within a pre-specified budget (Joseph, Maloney & Possingham 2009). Weights can be used to specify the relative importance of conserving specific features (see `add_feature_weights()`). Although this objective is conceptually similar to maximizing species richness (`add_max_richness_objective()` it is designed to work for features that are not associated with probabilistic outcomes.

Usage

```
add_max_wtd_sum_objective(x, budget)
```

Arguments

x	`problem()` object.
budget	`numeric` value representing the maximum amount of total expenditure for funding actions.

Details

A problem objective is used to specify the overall goal of the project prioritization problem. Here, this objective seeks to find the set of actions that maximizes the weighted sum of the expected outcomes (e.g., probability of persistence, population size, amount of habitat) associated with the features (e.g., populations, species, ecosystems) 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 expected outcome for each feature 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 outcome for each feature $f \in F$ associated with the project $j \in J$ assuming that all of the actions comprising project j are funded and that project is allocated to feature f. For convenience, let Q_{fj} denote the expected outcome for each $f \in F$ associated with the project $j \in J$ 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 expected outcome for a feature given that a project is funded and allocated to the feature depends on (i) the probability of the project succeeding, (ii) the

outcome for the feature if the project succeeds, and (iii) the outcome for the feature if the project fails (per the baseline project). 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; and the binary Z_{fj} variables indicate if project j is used to manage feature f or not.

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).

$$\text{Maximize} \sum_{f=0}^{F} E_f W_f \text{(eqn1a)} \text{Subject to} \sum_{i=0}^{I} C_i \leq m \text{(eqn1b)} E_f = \sum_{j=0}^{J} Z_{fj} Q_{fj} \forall f \in F \text{(eqn1c)} Z_{fj} \leq Y_j \forall j \in J \text{(eqn1d)} \sum_{j=}$$

The objective (eqn 1a) is to maximize the weighted sum of the expected outcome values for the features. 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 expected outcome for each feature 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 expected outcome variables (E_f) are greater than zero. Constraints (eqns 1h) ensure that the action funding (X_i), project funding (Y_j), and project allocation (Z_{fj}) variables are binary.

Value

A problem() 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

See objectives for an overview of functions for adding objectives.

Other objectives: add_max_phylo_div_objective(), add_max_richness_objective(), add_max_targets_met_object
add_min_set_objective()

Examples

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

# build problem with maximum weighted sum objective and $300 budget
p1 <-
  problem(
    sim_projects, sim_actions, sim_features,
```

```
    "name", "success", "name", "cost", "name"
  ) %>%
  add_max_wtd_sum_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_min_set_objective` *Add minimum set objective*

Description

Add an objective to a project prioritization problem based on minimizing the cost of the solution, whilst ensuring that the target thresholds for each feature is met (Chadés *et al.* 2015). In some project prioritization exercises, decision makers may have a target threshold level of expected outcome for each feature (e.g., a 90% chance of persistence for each feature). This objective is especially useful for identifying solutions that ensure that each feature achieves, at least, some minimum level of expected outcome based on the target thresholds.

Usage

```
add_min_set_objective(x)
```

Arguments

x	`problem()` 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 targets for the expected outcomes (e.g., probabilities of persistence, population sizes, amount of habitat) associated with 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 target for feature f, and E_f denote the expected outcome for each feature 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 outcome for each feature $f \in F$ associated with the project $j \in J$ assuming that all of the actions comprising project j are funded and that project is allocated to feature f. For convenience, let Q_{fj} denote the expected outcome for each $f \in F$ associated with the project $j \in J$ 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 expected outcome for a feature given that a project is funded and allocated to the feature depends on (i) the probability of the project succeeding, (ii) the outcome for the feature if the project succeeds, and (iii) the outcome for the feature if the project fails (per the baseline project). 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 continuous E_f variables denote the expected outcome for feature f.

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).

$$\text{Minimize} \sum_{i=0}^{I} C_i X_i (\text{eqn1a}) \text{Subject to} E_f \geq T_f \forall f \in F (\text{eqn1b}) E_f = \sum_{j=0}^{J} Z_{fj} Q_{fj} \forall f \in F (\text{eqn1c}) Z_{fj} \leq Y_j \forall j \in J (\text{eqn1d})$$

The objective (eqn 1a) is to minimize the cost of the funded actions. Constraints (eqn 1b) ensure that the targets are met. Constraints (eqn 1c) calculate the expected outcome for each feature 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 expected outcome variables (E_f) are greater than zero. Constraints (eqns 1h) ensure that the action funding (X_i), project funding (Y_j), and project allocation (Z_{fj}) variables are binary.

Value

A `problem()` with the objective added to it.

References

Chadés 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

See targets for an overview of functions for adding targets.

Other objectives: add_max_phylo_div_objective(), add_max_richness_objective(), add_max_targets_met_object
add_max_wtd_sum_objective()

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

Add a solver to generate solutions to a project prioritization problem based on random selection. Although prioritizations should be developed using optimization routines, a portfolio of randomly generated solutions can be useful for evaluating the effectiveness of an optimized solution.

Usage

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

Arguments

x	problem() or multi_problem() 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 during optimization? Defaults to TRUE.

Details

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

The following steps are used for objectives that maximize benefit subject to budgetary constraints (e.g., add_max_wtd_sum_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.

The following steps are used for objectives that 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

A problem() object with the solver added to it.

See Also

Other solvers: add_cbc_solver(), add_default_solver(), add_gurobi_solver(), add_heuristic_solver(), add_highs_solver(), add_lpsolveapi_solver(), add_lpsymphony_solver(), add_rsymphony_solver()

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_wtd_sum_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
# according to the weighted sum objective
hist(
  s1$obj,
  xlab = "Expected outcome", 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_ref_point_approach` *Add a reference point approach*

Description

Add a reference point approach for multi-objective optimization to a project problem (Vanderpooten 1990). Note that this function can only be applied when all objectives should be maximized.

Usage

```
add_ref_point_approach(
  x,
  weights,
  goals,
  best = NULL,
  worst = NULL,
  verbose = TRUE
)
```

Arguments

`x`	`multi_problem()` object.
`weights`	`numeric` vector containing the weights for each objective. To generate multiple solutions based on different values, `weights` can be a `numeric` matrix where each row corresponds to a different solution and each columns corresponds to a different objective.
`goals`	`numeric` vector containing values that denote the reference points. These points represent aspirational goals for each objective. To generate multiple solutions based on different values, `goals` can be a `numeric` matrix where each row corresponds to a different solution and each columns corresponds to a different objective. Note that all values must be greater than zero.
`best`	`numeric` vector containing values that denote the worst possible performance for each objective. If `NULL`, then these values are computed automatically.
`worst`	`numeric` vector containing values that denote the worst possible performance for each objective. If `NULL`, then these values are computed automatically.
`verbose`	`logical` should progress on generating solutions displayed? Defaults to `TRUE`.

Details

The reference point approach for multi-objective optimization involves creating a new objective that is calculated based on multiple objectives. In particular, the new objective uses weights to specify the relative importance of each individual objective, and goals to specify a threshold minimum level of performance for each objective (conceptually similar to target thresholds used in conservation planning). Given this, the reference point approach first involves minimizing the maximum goal shortfall (i.e., difference between goal and objective value, expressed as a percentage), and then subsequently minimizing the weighted sum of the goal shortfalls.

To describe this approach mathematically, we will define the following terminology. Let O denote the set of objectives (indexed by o). For each objective, let W_o denote the weight for each objective $o \in O$, G_o denote the goal for each objective $o \in O$, A_o denote the best objective value for each objective, B_o denote the worst objective value for each objective, and V_o denote the objective value for a candidate solution as measured based on each objective $o \in O$. After defining these terms, the approach is formulated with the following equation.

$$\text{Minimize} \max_{o=0}^{O} W_o \times \frac{1}{B_o - W_o} \times \max(G_o - V_o, 0), \text{Minimize} \sum_{o=0}^{O} W_o \times \frac{1}{B_o - W_o} \times \max(G_o - V_o, 0)$$

Value

A `multi_problem()` object with the approach added to it.

References

Vanderpooten D (1990) *Multiobjective programming: Basic concepts and approaches*. In: Stochastic Versus Fuzzy Approaches to Multiobjective Mathematical Programming Under Uncertainty. Springer, Berlin.

See Also

See approaches for an overview of functions for adding approaches.

Other approaches: `add_abs_constraint_approach()`, `add_wtd_goal_approach()`

Examples

```
# load data
data(sim_multi_projects)
data(sim_multi_features)
data(sim_multi_actions)
data(sim_multi_tree)

# build problem
p <-
  multi_problem(
    obj1 =
      problem(
        sim_multi_projects[[1]], sim_multi_actions, sim_multi_features[[1]],
        "name", "success", "name", "cost", "name",
```

```
      baseline_project_name = "baseline_project_obj1"
    ) %>%
    add_max_phylo_div_objective(
      budget = 1000, tree = sim_multi_tree[[1]]
    ) %>%
    add_binary_decisions(),
  obj2 =
    problem(
      sim_multi_projects[[2]], sim_multi_actions, sim_multi_features[[2]],
      "name", "success", "name", "cost", "name",
      baseline_project_name = "baseline_project_obj2"
    ) %>%
    add_max_richness_objective(budget = 1000) %>%
    add_binary_decisions(),
  obj3 =
    problem(
      sim_multi_projects[[3]], sim_multi_actions, sim_multi_features[[3]],
      "name", "success", "name", "cost", "name",
      baseline_project_name = "baseline_project_obj3"
    ) %>%
    add_max_wtd_sum_objective(budget = 1000) %>%
    add_binary_decisions()
) %>%
add_ref_point_approach(weights = c(10, 11, 12), goals = c(3, 4, 5)) %>%
add_default_solver()

# print problem
print(p)

# solve problem
s <- solve(p)

# print solution
print(s)
```

`add_relative_targets` *Add relative targets*

Description

Add targets to a project prioritization problem that specify the desired expected outcome for each feature as a proportion of the the best possible outcome that could be achieved. For instance, if a feature is associated with three projects that would be expected to result in a 30%, 60%, and 80% chance of persistence (adjusted for baseline outcomes), then setting a relative target of 0.5 would correspond to a 40% chance of persistence (i.e., 50% times 80%).

Usage

```
add_relative_targets(x, targets)
```

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

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

Arguments

x	[problem() object.
targets	Object that specifies the targets for each feature. See the Details section for more information.

Details

Targets are used to specify a threshold minimum desirable expected outcome for each feature. These should ideally be set according to stakeholder requirements and expert knowledge. 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 using the following options.

`numeric` **value** The value is used to set the target threshold for each feature. This option may be useful when all features should be assigned the same target threshold.

`numeric` **vector** Each value specifies a target threshold for each feature. The order of the values should correspond to the order of the features in `x`.

`character` **value** The value specifies the name of a column in the feature data (i.e., the argument to `features` in the `problem()` function). The target threshold for each feature is set according the column values.

Value

A `problem()` object with the targets added to it.

See Also

Other targets: `add_absolute_targets()`, `add_manual_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

Add a solver to generate solutions to a project prioritization problem with the *SYMPHONY* software via the **Rsymphony** package. This function can also be used to customize the behavior of the solver. It requires the **Rsymphony** package to be installed.

Usage

```
add_rsymphony_solver(
  x,
  gap = 0,
  time_limit = .Machine$integer.max,
  first_feasible = FALSE,
  verbose = TRUE
)
```

Arguments

x	`problem()` or `multi_problem()` object.
gap	`numeric` gap to optimality. This gap is relative and expresses the acceptable deviance from the optimal objective. For example, a value of 0.01 will result in the solver stopping when it has found a solution within 1% of optimality. Additionally, a value of 0 will result in the solver stopping when it has found an optimal solution. The default value is 0 (i.e., 0% from optimality).
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 during optimization? Defaults to `TRUE`.

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

A problem() object with the solver added to it.

See Also

Other solvers: add_cbc_solver(), add_default_solver(), add_gurobi_solver(), add_heuristic_solver(), add_highs_solver(), add_lpsolveapi_solver(), add_lpsymphony_solver(), add_random_solver()

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_wtd_sum_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)
```

---

add_wtd_goal_approach *Add a weighted goal achievement approach*

Description

Add a weighted goal achievement approach for multi-objective optimization to a project problem (Jones and Tamiz 2010).

Usage

```
add_wtd_goal_approach(x, weights, goals, verbose = TRUE)
```

Arguments

x	multi_problem() object.
weights	numeric vector containing the weights for each objective. To generate multiple solutions based on different values, weights can be a numeric matrix where each row corresponds to a different solution and each columns corresponds to a different objective.
goals	numeric vector containing values that denote the reference points. These points represent aspirational goals for each objective. To generate multiple solutions based on different values, goals can be a numeric matrix where each row corresponds to a different solution and each columns corresponds to a different objective. Note that all values must be greater than zero.
verbose	logical should progress on generating solutions displayed? Defaults to TRUE.

Details

The weighted goal achievement approach for multi-objective optimization involves creating a new objective that is calculated based on multiple objectives. In particular, the new objective uses weights to specify the relative importance of each individual objective, and goals to specify a threshold minimum level of performance for each objective (conceptually similar to target thresholds used in conservation planning). It then calculates the new objectives based on the weighted sum of the percentage of each goal that is achieved for each objective.

To describe this approach mathematically, we will define the following terminology. Let O denote the set of objectives (indexed by o). For each objective, let W_o denote the weight for each objective $o \in O$, G_o denote the goal for each objective $o \in O$, and V_o denote the objective value for a candidate solution as measured based on each objective $o \in O$. After defining these terms, the approach is formulated with the following equation.

$$\text{Minimize} \sum_{o=0}^{O} W_o \times \frac{V_o}{G_o}$$

Value

A multi_problem() object with the approach added to it.

References

Jones D and Tamiz M (2010) *Goal Programming Variants*. and Management Science, volume 141. Springer, Boston, MA.

See Also

Other approaches: add_abs_constraint_approach(), add_ref_point_approach()

Examples

```
# load data
data(sim_multi_projects)
data(sim_multi_features)
data(sim_multi_actions)
data(sim_multi_tree)

# build problem
p <-
  multi_problem(
    obj1 =
      problem(
        sim_multi_projects[[1]], sim_multi_actions, sim_multi_features[[1]],
        "name", "success", "name", "cost", "name",
        baseline_project_name = "baseline_project_obj1"
      ) %>%
      add_max_phylo_div_objective(
        budget = 1000, tree = sim_multi_tree[[1]]
      ) %>%
      add_binary_decisions(),
   obj2 =
     problem(
       sim_multi_projects[[2]], sim_multi_actions, sim_multi_features[[2]],
       "name", "success", "name", "cost", "name",
       baseline_project_name = "baseline_project_obj2"
     ) %>%
     add_max_richness_objective(budget = 1000) %>%
     add_binary_decisions(),
   obj3 =
     problem(
       sim_multi_projects[[3]], sim_multi_actions, sim_multi_features[[3]],
       "name", "success", "name", "cost", "name",
       baseline_project_name = "baseline_project_obj3"
     ) %>%
     add_max_wtd_sum_objective(budget = 1000) %>%
     add_binary_decisions()
 ) %>%
 add_wtd_goal_approach(weights = c(10, 11, 12), goals = c(3, 4, 5)) %>%
 add_default_solver()

# print problem
print(p)

# solve problem
s <- solve(p)

# print solution
print(s)
```

approaches *Multi-objective optimization approaches*

Description

Approaches specify methods for generating solutions for multi-objective optimization problems.

Details

The following approaches can be used to generate solutions for a multi-objective project prioritization problem.

`add_abs_constraint_approach()` Add an approach to generate solutions based on constraints that specify the required objectives values.

`add_wtd_goal_approach()` Add an approach to generate solutions with the weighted goal method (Jones and Tamiz 2010).

`add_ref_point_approach()` Add an approach to generate solutions with the reference point method (Vanderpooten 1990)

References

Jones D and Tamiz M (2010) *Goal Programming Variants*. and Management Science, volume 141. Springer, Boston, MA.

Vanderpooten D (1990) *Multiobjective programming: Basic concepts and approaches*. In: Stochastic Versus Fuzzy Approaches to Multiobjective Mathematical Programming Under Uncertainty. Springer, Berlin.

See Also

Other overviews: `constraints`, `objectives`, `solvers`, `targets`, `weights`

Examples

```
# load data
data(sim_multi_projects)
data(sim_multi_features)
data(sim_multi_actions)
data(sim_multi_tree)

# build problem
p1 <-
  multi_problem(
    obj1 =
      problem(
        sim_multi_projects[[1]], sim_multi_actions, sim_multi_features[[1]],
        "name", "success", "name", "cost", "name",
        baseline_project_name = "baseline_project_obj1"
      ) %>%
```

```
      add_max_phylo_div_objective(
       budget = 1000, tree = sim_multi_tree[[1]]
      ) %>%
      add_binary_decisions(),
    obj2 =
      problem(
        sim_multi_projects[[2]], sim_multi_actions, sim_multi_features[[2]],
        "name", "success", "name", "cost", "name",
        baseline_project_name = "baseline_project_obj2"
      ) %>%
      add_max_richness_objective(budget = 1000) %>%
      add_binary_decisions(),
    obj3 =
      problem(
        sim_multi_projects[[3]], sim_multi_actions, sim_multi_features[[3]],
        "name", "success", "name", "cost", "name",
        baseline_project_name = "baseline_project_obj3"
      ) %>%
      add_max_wtd_sum_objective(budget = 1000) %>%
      add_binary_decisions()
 ) %>%
 add_default_solver()

# build another problem, with the absolute constraint method
p2 <-
  p1 %>%
  add_abs_constraint_approach(
    goals = c(NA, 0.01, 0.01)
  )

# build another problem, with the weighted goal method
p3 <-
  p1 %>%
  add_wtd_goal_approach(
    weights = c(1, 0.5, 0.1),
    goals = c(1, 3, 0.2)
  )

# build another problem, with the reference point method
p4 <-
  p1 %>%
  add_ref_point_approach(
    weights = c(1, 0.5, 0.1),
    goals = c(1, 3, 0.2)
  )

# generate solutions using each approach
s <- rbind(solve(p2), solve(p3), solve(p4))
s$approach <- c("abs epsilon", "wtd goal", "ref point")

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

as.list.OptimizationProblem *Convert* OptimizationProblem *to list*

Description

Convert OptimizationProblem to list

Usage

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

Arguments

x	OptimizationProblem 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	ape::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

A Matrix::dgCMatrix 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")
```

---

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, n_approx = 100, ...)

## S3 method for class 'MultiObjProjectProblem'
multi_compile(x, ...)

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

Arguments

x	ProjectProblem object.
...	not used.
n_approx	integer number of points to use for piece-wise linear approximations of non-linear terms. Defaults to 100.

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

An OptimizationProblem object.

Examples

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

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

# print problem
print(p)

# compile problem
o <- compile(p)

# print compiled problem
print(o)
```

Constraint-class *Constraint class*

Description

This class is used to represent the constraints used in optimization. **Only experts should use the fields and methods for this class directly.**

Super class

ProjectModifier -> Constraint

Methods

Public methods:

- Constraint$clone()

Constraint$clone(): The objects of this class are cloneable with this method.

Usage:

```
Constraint$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other classes: Decision-class, MultiObjApproach-class, MultiObjProjectProblem-class, Objective-class, OptimizationProblem-class, ProjectModifier-class, ProjectProblem-class, Solver-class, Target-class, Weight-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_action_constraints() Add constraints to ensure that particular actions are selected for funding.

add_locked_out_action_constraints() Add constraints to ensure that particular actions are not selected for funding.

add_manual_locked_action_constraints() Add constraints to ensure that particular actions are selected, or not, for funding.

add_locked_in_project_constraints() Add constraints to ensure that particular projects are selected for funding.

add_locked_out_project_constraints() Add constraints to ensure that particular projects are not selected for funding.

add_manual_locked_project_constraints() Add constraints to ensure that particular projects are selected, or not, for funding.

See Also

Other overviews: approaches, objectives, solvers, targets, weights

Examples

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

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

# print problem
print(p1)

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

# print problem
print(p2)

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

# print problem
print(p3)

# build another problem, and lock in the first project
p4 <- p1 %>% add_locked_out_project_constraints(c(1))

# print problem
print(p4)

# build another problem, and lock out the second project
p5 <- p1 %>% add_locked_out_project_constraints(c(2))

# print problem
print(p5)

# solve problems
s1 <- solve(p1)
s2 <- solve(p2)
s3 <- solve(p3)
s4 <- solve(p4)
s5 <- solve(p5)

# print the projects 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])
print(s5[, sim_actions$name])
```

`Decision-class` *Decision class*

Description

This class is used to represent the decision variables used in optimization. **Only experts should use the fields and methods for this class directly.**

Super class

`ProjectModifier` -> `Decision`

Methods

Public methods:

- `Decision$clone()`

`Decision$clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Decision$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other classes: `Constraint-class`, `MultiObjApproach-class`, `MultiObjProjectProblem-class`, `Objective-class`, `OptimizationProblem-class`, `ProjectModifier-class`, `ProjectProblem-class`, `Solver-class`, `Target-class`, `Weight-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 weighted sum objective, $200 budget, and
# binary decisions
p <-
  problem(
    sim_projects, sim_actions, sim_features,
    "name", "success", "name", "cost", "name"
  ) %>%
  add_max_wtd_sum_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

Get the names of the features in an object.

Usage

```
feature_names(x)

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

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

Arguments

x	problem() or multi_problem() object.

Value

A character vector or list of character vectors.

Examples

```
# 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_wtd_sum_objective(budget = 200) %>%
  add_binary_decisions() %>%
  add_default_solver()

# print problem
print(p)

# print feature names
feature_names(p)
```

is.Waiver *Is waiver?*

Description

Check if an object is a `waiver`.

Usage

```
is.Waiver(x)
```

Arguments

`x`	Object.

Value

A `logical` value.

Examples

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

# is it a waiver object?
is.Waiver(w)
```

MultiObjApproach-class *Multi-objective approach class*

Description

This class is used to represent approaches for multi-objective optimization. **Only experts should use the fields and methods for this class directly.**

Super class

`ProjectModifier` -> `MultiObjApproach`

Methods

Public methods:

- `MultiObjApproach$calculate()`
- `MultiObjApproach$run()`
- `MultiObjApproach$clone()`

`MultiObjApproach$calculate()`: Perform computations that need to be completed before applying the object.

Usage:

```
MultiObjApproach$calculate(x, y)
```

Arguments:

`x` `list` containing a compiled multi-objective optimization problem (e.g., generated with internal function `multi_compile()`).

`y` `multi_problem()` object.

Returns: Invisible `TRUE`.

`MultiObjApproach$run()`: Solve a multi-objective optimization problem to generate a solution.

Usage:

```
MultiObjApproach$run(x)
```

Arguments:

`x` `list` containing a compiled multi-objective optimization problem (e.g., generated with internal function `multi_compile()`).

Returns: `list` of solutions.

`MultiObjApproach$clone()`: The objects of this class are cloneable with this method.

Usage:

```
MultiObjApproach$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other classes: `Constraint-class`, `Decision-class`, `MultiObjProjectProblem-class`, `Objective-class`, `OptimizationProblem-class`, `ProjectModifier-class`, `ProjectProblem-class`, `Solver-class`, `Target-class`, `Weight-class`

`MultiObjProjectProblem-class`

Multi-objective project problem class

Description

This class is used to represent multi-objective project planning problems. It stores the data (e.g., actions, and features) and mathematical formulation (e.g., the objective, constraints, and other design criteria) needed to generate prioritizations. Most users should use `multi_problem()` to generate new multi-objective project problem objects, and the functions distributed with the package to interact with them. **Only experts should use the fields and methods for this class directly.**

Public fields

`problems` `list` containing `ProjectProblem` objects.

`defaults` `list` indicating if other fields contain defaults.

`approach` `MultiObjApproach` object for specifying the multi-objective optimization approach.

`solver` `Solver` object specifying the solver for generating solutions.

Methods

Public methods:

- `MultiObjProjectProblem$new()`
- `MultiObjProjectProblem$print()`
- `MultiObjProjectProblem$show()`
- `MultiObjProjectProblem$repr()`
- `MultiObjProjectProblem$number_of_problems()`
- `MultiObjProjectProblem$number_of_features()`
- `MultiObjProjectProblem$number_of_actions()`
- `MultiObjProjectProblem$number_of_projects()`
- `MultiObjProjectProblem$problem_names()`
- `MultiObjProjectProblem$feature_names()`
- `MultiObjProjectProblem$action_names()`
- `MultiObjProjectProblem$project_names()`
- `MultiObjProjectProblem$add_approach()`
- `MultiObjProjectProblem$add_solver()`
- `MultiObjProjectProblem$clone()`

`MultiObjProjectProblem$new()`**:** Create a new multi-objective conservation problem object.

Usage:

```
MultiObjProjectProblem$new(problems)
```

Arguments:

`problems` `list` containing `ProjectProblem` objects.

Returns: A new `MultiObjProjectProblem` object.

`MultiObjProjectProblem$print()`: Print concise information about the object.

Usage:

```
MultiObjProjectProblem$print()
```

Returns: Invisible `TRUE`.

`MultiObjProjectProblem$show()`: Display concise information about the object.

Usage:

```
MultiObjProjectProblem$show()
```

Returns: Invisible `TRUE`.

`MultiObjProjectProblem$repr()`: Generate a character representation of the object.

Usage:

```
MultiObjProjectProblem$repr()
```

Returns: A `character` value.

`MultiObjProjectProblem$number_of_problems()`: Obtain the number of problems.

Usage:

```
MultiObjProjectProblem$number_of_problems()
```

Returns: An `integer` value.

`MultiObjProjectProblem$number_of_features()`: Obtain the number of features.

Usage:

```
MultiObjProjectProblem$number_of_features()
```

Returns: An `integer` value.

`MultiObjProjectProblem$number_of_actions()`: Obtain the number of actions.

Usage:

```
MultiObjProjectProblem$number_of_actions()
```

Returns: An `integer` value.

`MultiObjProjectProblem$number_of_projects()`: Obtain the number of projects.

Usage:

```
MultiObjProjectProblem$number_of_projects()
```

Returns: An `integer` value.

`MultiObjProjectProblem$problem_names()`: Obtain the names of the problems.

Usage:

```
MultiObjProjectProblem$problem_names()
```

Returns: An `character` value.

`MultiObjProjectProblem$feature_names()`: Obtain the names of the features.

Usage:

```
MultiObjProjectProblem$feature_names()
```

Returns: A `list` of `character` vectors.

`MultiObjProjectProblem$action_names()`: Obtain the names of the actions.

Usage:

```
MultiObjProjectProblem$action_names()
```

Returns: A `character` vector.

`MultiObjProjectProblem$project_names()`: Obtain the names of the projects.

Usage:

```
MultiObjProjectProblem$project_names()
```

Returns: A `list` of `character` vectors.

`MultiObjProjectProblem$add_approach()`: Create a new object with an approach added to the problem formulation.

Usage:

```
MultiObjProjectProblem$add_approach(x)
```

Arguments:

`x` MultiObjApproach object.

Returns: An updated `MultiObjProjectProblem` object.

`MultiObjProjectProblem$add_solver()`: Create a new object with a solver added to the problem formulation.

Usage:

```
MultiObjProjectProblem$add_solver(x)
```

Arguments:

`x` Solver object.

Returns: An updated `MultiObjProjectProblem` object.

`MultiObjProjectProblem$clone()`: The objects of this class are cloneable with this method.

Usage:

```
MultiObjProjectProblem$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other classes: `Constraint-class`, `Decision-class`, `MultiObjApproach-class`, `Objective-class`, `OptimizationProblem-class`, `ProjectModifier-class`, `ProjectProblem-class`, `Solver-class`, `Target-class`, `Weight-class`

`multi_problem` *Multi-objective project prioritization problem*

Description

Create a multi-objective systematic project prioritization problem.

Usage

```
multi_problem(..., problem_names = NULL)
```

Arguments

`...`	`problem()` objects.
`problem_names`	`character` vector with a name for each problem in `...`. Defaults to `NULL`, such that the problem names are defined automatically.

Details

A multi-objective project prioritization problem contains multiple single-objective project prioritization problems (i.e., created with `problem()`). Each of these single-objective project prioritization problems must have exactly the same actions (i.e., argument to `actions`). Additionally, each single-objective project prioritization problem must have a different set of projects and features (i.e., they have have different names).

Value

A MultiObjProjectProblem object.

Examples

```
# load data
data(sim_multi_projects)
data(sim_multi_features)
data(sim_multi_actions)
data(sim_multi_tree)

# build problem
p <-
  multi_problem(
    obj1 =
      problem(
        sim_multi_projects[[1]], sim_multi_actions, sim_multi_features[[1]],
        "name", "success", "name", "cost", "name",
        baseline_project_name = "baseline_project_obj1"
      ) %>%
      add_max_phylo_div_objective(
       budget = 200, tree = sim_multi_tree[[1]]
      ) %>%
```

```
    add_binary_decisions(),
  obj2 =
    problem(
      sim_multi_projects[[2]], sim_multi_actions, sim_multi_features[[2]],
      "name", "success", "name", "cost", "name",
      baseline_project_name = "baseline_project_obj2"
    ) %>%
    add_max_richness_objective(budget = 200) %>%
    add_binary_decisions(),
  obj3 =
    problem(
      sim_multi_projects[[3]], sim_multi_actions, sim_multi_features[[3]],
      "name", "success", "name", "cost", "name",
      baseline_project_name = "baseline_project_obj3"
    ) %>%
    add_max_wtd_sum_objective(budget = 200) %>%
    add_binary_decisions()
) %>%
add_ref_point_approach(weights = c(10, 11, 12), goals = c(3, 4, 5)) %>%
add_default_solver()

# print problem
print(p)

# solve problem
s <- solve(p)

# print solution
print(s)
```

`new_optimization_problem` *Optimization problem*

Description

Generate a new empty OptimizationProblem object.

Usage

```
new_optimization_problem()
```

Value

An OptimizationProblem object.

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

An 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

Get 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 'MultiObjProjectProblem'
number_of_actions(x)
```

Arguments

x	problem() object.

Value

An integer value.

Examples

```
# 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_wtd_sum_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

Get 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 'MultiObjProjectProblem'
number_of_features(x)
```

Arguments

x	problem() object.

Value

An integer value.

Examples

```
# 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_wtd_sum_objective(budget = 200) %>%
  add_binary_decisions() %>%
  add_default_solver()

# print problem
print(p)

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

number_of_problems *Number of problems*

Description

Get the number of problems in an object.

Usage

```
number_of_problems(x)

## S4 method for signature 'MultiObjProjectProblem'
number_of_problems(x)
```

Arguments

x	problem() object.

Value

An integer value.

Examples

```
# load data
data(sim_multi_projects)
data(sim_multi_features)
data(sim_multi_actions)
data(sim_multi_tree)

# build problem
p <-
  multi_problem(
    obj1 =
      problem(
        sim_multi_projects[[1]], sim_multi_actions, sim_multi_features[[1]],
        "name", "success", "name", "cost", "name",
        baseline_project_name = "baseline_project_obj1"
      ) %>%
      add_max_phylo_div_objective(
        budget = 200, tree = sim_multi_tree[[1]]
      ) %>%
      add_binary_decisions(),
    obj2 =
      problem(
        sim_multi_projects[[2]], sim_multi_actions, sim_multi_features[[2]],
        "name", "success", "name", "cost", "name",
        baseline_project_name = "baseline_project_obj2"
      ) %>%
```

```
    add_max_richness_objective(budget = 200) %>%
    add_binary_decisions(),
  obj3 =
    problem(
      sim_multi_projects[[3]], sim_multi_actions, sim_multi_features[[3]],
      "name", "success", "name", "cost", "name",
      baseline_project_name = "baseline_project_obj3"
    ) %>%
    add_max_wtd_sum_objective(budget = 200) %>%
    add_binary_decisions()
)

# print problem
print(p)

# print number of problems
number_of_problems(p)
```

`number_of_projects` *Number of projects*

Description

Get 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 'MultiObjProjectProblem'
number_of_projects(x)
```

Arguments

x	`problem()` object.

Value

An `integer` value.

Examples

```
# 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_wtd_sum_objective(budget = 200) %>%
  add_binary_decisions() %>%
  add_default_solver()

# print problem
print(p)

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

`Objective-class` *Objective class*

Description

This class is used to represent the objective function used in optimization. **Only experts should use the fields and methods for this class directly.**

Super class

`ProjectModifier` -> `Objective`

Public fields

`has_targets` `logical` value indicating if the objective uses targets.

`has_weights` `logical` value indicating if the objective uses weights.

Methods

Public methods:

- `Objective$feature_phylogeny()`
- `Objective$default_feature_weights()`
- `Objective$replace_feature_weights()`
- `Objective$evaluate()`
- `Objective$clone()`

`Objective$feature_phylogeny()`: Obtain the feature phylogeny.

Usage:

```
Objective$feature_phylogeny()
```

Returns: A `ape::phylo()` phylogenetic tree object.

`Objective$default_feature_weights()`: Obtain default feature weights.

Usage:

```
Objective$default_feature_weights()
```

Returns: A `numeric` vector with the default feature weights.

`Objective$replace_feature_weights()`**:** Should default feature weights be replaced or multiplied by the new weights?

Usage:

```
Objective$replace_feature_weights()
```

Returns: A `logical` value.ks

`Objective$evaluate()`**:** Calculate the objective value for a solution.

Usage:

```
Objective$evaluate(y, solution)
```

Arguments:

`y` ProjectProblem object.

`solution` `tibble::tibble()` object with solution.

Returns: A `numeric` value.

`Objective$clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Objective$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other classes: `Constraint-class`, `Decision-class`, `MultiObjApproach-class`, `MultiObjProjectProblem-class`, `OptimizationProblem-class`, `ProjectModifier-class`, `ProjectProblem-class`, `Solver-class`, `Target-class`, `Weight-class`

`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 expected to persist into the future (Joseph, Maloney & Possingham 2009).

`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_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_min_set_objective()` Minimize the cost of the solution, whilst ensuring that all targets are met (Chadés *et al.* 2015),

`add_max_wtd_sum_objective()` Maximize the weighted sum of the expected outcomes of the features, whilst ensuring that the cost of the solution is within a pre-specified budget (Joseph, Maloney & Possingham 2009).

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.

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

Other overviews: `approaches`, `constraints`, `solvers`, `targets`, `weights`

Examples

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

# build problem
p1 <-
  problem(
    sim_projects, sim_actions, sim_features,
    "name", "success", "name", "cost", "name"
  ) %>%
```

```
  add_binary_decisions()

# build problem with maximum richness objective and $200 budget
p2 <- p1 %>% add_max_richness_objective(budget = 200)

# build problem with maximum phylogenetic diversity objective and $200 budget
p3 <- p1 %>% add_max_phylo_div_objective(budget = 200, tree = sim_tree)

# build problem with maximum targets met objective, $200 budget, and
# 40% persistence targets
p4 <-
  p1 %>%
  add_max_targets_met_objective(budget = 200) %>%
  add_absolute_targets(0.4)

# build problem with minimum set objective and 40% persistence targets
p5 <-
  p1 %>%
  add_min_set_objective() %>%
  add_absolute_targets(0.4)

# build problem with maximum weighted sum objective and $200 budget,
# note that this is identical to the maximum richness objective
# when using probability of persistence values, such as those
# present in the simulated data
p6 <- p1 %>% add_max_wtd_sum_objective(budget = 200)

# solve problems
s <- rbind(solve(p2), solve(p3), solve(p4), solve(p5), solve(p6))

# print solutions
print(s)
```

---

oppr *oppr: Optimal Project Prioritization*

Description

The **oppr** package a decision support tool for prioritizing conservation projects. Prioritizations can be developed by maximizing expected outcomes as a weighted sum (e.g., species 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.

Details

This package has a vignette to showcase its usage. To view the vignette, please use the code `vignette("oppr", package = "oppr")`.

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 `https://support.gurobi.com/hc/en-us/articles/4534161999889-How-do-I-install-Gurobi-Optimizer` for instructions). 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 `https://support.gurobi.com/hc/en-us/articles/14462206790033-How-do-I-install-Gurobi-for-R` for instructions).

Citation

Please cite the *oppr R* package when using it in publications. To cite the package, please use:

Hanson JO, Schuster R, Strimas-Mackey M & Bennett JR (2019) Optimality in prioritizing conservation projects. *Methods in Ecology & Evolution*, 10: 1655–1663.

Author(s)

Authors:

- Jeffrey O Hanson <jeffrey.hanson@uqconnect.edu.au> (ORCID)
- Richard Schuster <richard.schuster@glel.carleton.ca> (ORCID, maintainer)
- Matthew Strimas-Mackey <mstrimas@gmail.com> (ORCID)
- Joseph Bennett <joseph.bennett@carleton.ca> (ORCID)

See Also

Useful links:

- Package website (`https://prioritizr.github.io/oppr/`)
- Source code repository (`https://github.com/prioritizr/oppr`)
- Report bugs (`https://github.com/prioritizr/oppr/issues`)

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_wtd_sum_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)
```

`oppr-deprecated` *Deprecation notice*

Description

The functions listed here are deprecated. This means that they once existed in earlier versions of the of the **oppr** package, but they have since been removed entirely, replaced by other functions, or renamed as other functions in newer versions. To help make it easier to transition to new versions of the **oppr** package, we have listed alternatives for deprecated the functions (where applicable). If a function is described as being renamed, then this means that only the name of the function has changed (i.e., the inputs, outputs, and underlying code remain the same).

Usage

```
plot_feature_persistence(...)

plot_phylo_persistence(...)

add_locked_in_constraints(...)

add_locked_out_constraints(...)

add_manual_locked_constraints(...)
```

Arguments

...	not used.

Details

The following functions have been deprecated:

`add_max_richness_objective()` renamed as the `add_max_wtd_sum_objective()` function.

`plot_feature_persistence()` renamed as the `plot_solution_barplot()` function.

`plot_phylo_persistence()` renamed as the `plot_solution_phylogram()` function.

`add_locked_in_constraints()` renamed as the `add_locked_in_action_constraints()` function.

`add_locked_out_constraints()` renamed as the `add_locked_out_action_constraints()` function.

`add_manual_locked_constraints()` renamed as the `add_manual_locked_action_constraints()` function.

OptimizationProblem-class

Optimization problem class

Description

This class is used to represent an optimization problem. It stores the information needed to generate a solution using an exact algorithm solver. Most users should use `compile()` to generate new optimization problem objects, and the functions distributed with the package to interact with them (e.g., `base::as.list()`). **Only experts should use the fields and methods for this class directly.**

Public fields

`ptr` A `Rcpp::Xptr` external pointer.

`data` A `list` with supplemental data. Create a new optimization problem object.

Methods

Public methods:

- OptimizationProblem$new()
- OptimizationProblem$get_data()
- OptimizationProblem$print()
- OptimizationProblem$show()
- OptimizationProblem$ncol()
- OptimizationProblem$nrow()
- OptimizationProblem$ncell()
- OptimizationProblem$modelsense()
- OptimizationProblem$vtype()
- OptimizationProblem$obj()
- OptimizationProblem$pwlobj()
- OptimizationProblem$A()
- OptimizationProblem$rhs()
- OptimizationProblem$sense()
- OptimizationProblem$lb()
- OptimizationProblem$ub()
- OptimizationProblem$number_of_features()
- OptimizationProblem$number_of_branches()
- OptimizationProblem$number_of_allocations()
- OptimizationProblem$number_of_actions()
- OptimizationProblem$number_of_projects()
- OptimizationProblem$col_ids()
- OptimizationProblem$row_ids()
- OptimizationProblem$copy()
- OptimizationProblem$convert_pwlobj()
- OptimizationProblem$clone()

`OptimizationProblem$new()`:

Usage:

```
OptimizationProblem$new(ptr, data = list())
```

Arguments:

`ptr` `Rcpp::Xptr` external pointer.
`data` `list` with supplemental data.

Returns: A new `OptimizationProblem` object.

`OptimizationProblem$get_data()`: Obtain the supplemental data.

Usage:

```
OptimizationProblem$get_data()
```

Returns: A `list` object.

`OptimizationProblem$print()`: Print concise information about the object.

Usage:

```
OptimizationProblem$print()
```

Returns: Invisible `TRUE`.

`OptimizationProblem$show()`: Print concise information about the object.

Usage:

```
OptimizationProblem$show()
```

Returns: Invisible `TRUE`.

`OptimizationProblem$ncol()`: Obtain the number of columns in the problem formulation.

Usage:

```
OptimizationProblem$ncol()
```

Returns: A `numeric` value.

`OptimizationProblem$nrow()`: Obtain the number of rows in the problem formulation.

Usage:

```
OptimizationProblem$nrow()
```

Returns: A `numeric` value.

`OptimizationProblem$ncell()`: Obtain the number of cells in the problem formulation.

Usage:

```
OptimizationProblem$ncell()
```

Returns: A `numeric` value.

`OptimizationProblem$modelsense()`: Obtain the model sense.

Usage:

```
OptimizationProblem$modelsense()
```

Returns: A `character` value.

`OptimizationProblem$vtype()`: Obtain the decision variable types.

Usage:

```
OptimizationProblem$vtype()
```

Returns: A `character` vector.

`OptimizationProblem$obj()`: Obtain the objective function.

Usage:

```
OptimizationProblem$obj()
```

Returns: A `numeric` vector.

`OptimizationProblem$pwlobj()`: Obtain the piecewise linear components of the objective function.

Usage:

```
OptimizationProblem$pwlobj()
```

Returns: A `list` object.

`OptimizationProblem$A()`: Obtain the constraint matrix.

Usage:

```
OptimizationProblem$A()
```

Returns: A `Matrix::sparseMatrix()` object.

`OptimizationProblem$rhs()`: Obtain the right-hand-side constraint values.

Usage:

```
OptimizationProblem$rhs()
```

Returns: A `numeric` vector.

`OptimizationProblem$sense()`: Obtain the constraint senses.

Usage:

```
OptimizationProblem$sense()
```

Returns: A `character` vector.

`OptimizationProblem$lb()`: Obtain the lower bounds for the decision variables.

Usage:

```
OptimizationProblem$lb()
```

Returns: A `numeric` vector.

`OptimizationProblem$ub()`: Obtain the upper bounds for the decision variables.

Usage:

```
OptimizationProblem$ub()
```

Returns: A `numeric` vector.

`OptimizationProblem$number_of_features()`: Obtain the number of features.

Usage:

```
OptimizationProblem$number_of_features()
```

Returns: A `numeric` value.

`OptimizationProblem$number_of_branches()`: Obtain the number of phylogenetic branches.

Usage:

```
OptimizationProblem$number_of_branches()
```

Returns: A `numeric` value.

`OptimizationProblem$number_of_allocations()`: Obtain the number of allocation variables. This number represents the total number of decision variables used to identify if each project is allocated to each variable.

Usage:

```
OptimizationProblem$number_of_allocations()
```

Returns: A `numeric` value.

`OptimizationProblem$number_of_actions()`: Obtain the number of actions

Usage:

```
OptimizationProblem$number_of_actions()
```

Returns: A `numeric` value.

`OptimizationProblem$number_of_projects()`: Obtain the number of projects.

Usage:

```
OptimizationProblem$number_of_projects()
```

Returns: A `numeric` value.

`OptimizationProblem$col_ids()`: Obtain the identifiers for the columns.

Usage:

```
OptimizationProblem$col_ids()
```

Returns: A `character` value.

`OptimizationProblem$row_ids()`: Obtain the identifiers for the rows.

Usage:

```
OptimizationProblem$row_ids()
```

Returns: A `character` value.

`OptimizationProblem$copy()`: Copy the object.

Usage:

```
OptimizationProblem$copy()
```

Returns: An `OptimizationProblem` object.

`OptimizationProblem$convert_pwlobj()`: Convert the piece-wise linear components of the objective function into linear objective components and constraints.

Usage:

```
OptimizationProblem$convert_pwlobj()
```

Returns: An invisible `TRUE`.

`OptimizationProblem$clone()`: The objects of this class are cloneable with this method.

Usage:

```
OptimizationProblem$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other classes: `Constraint-class`, `Decision-class`, `MultiObjApproach-class`, `MultiObjProjectProblem-class`, `Objective-class`, `ProjectModifier-class`, `ProjectProblem-class`, `Solver-class`, `Target-class`, `Weight-class`

`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`	`problem()` or `multi_problem()` object.
`solution`	`base::data.frame()` or `tibble::tibble()` 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, though 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 based on the projects selected for funding by the solution. 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 the expected outcome for each feature based on the projects selected for funding by the solution, and the color of the bars indicate each feature's weight. Additionally, regardless of the problem objective, 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 ggplot2::ggplot() object. If return_data = TRUE, then a data.frame is returned.

See Also

This function is a wrapper for plot_solution_phylogram() and plot_solution_barplot(), 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_wtd_sum_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_solution_barplot *Plot a bar plot to visualize a project prioritization*

Description

Create a bar plot to visualize the expected outcome associated with each feature given the projects selected for funding by a solution.

Usage

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

Arguments

x	problem() or multi_problem() object.
solution	base::data.frame() or tibble::tibble() 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, though 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 expected outcome for the a given feature (e.g., probability of persistence), 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 ggplot2::ggplot() object. If return_data = TRUE, then a data.frame is returned.

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_wtd_sum_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_solution_phylogram

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

Arguments

x	problem() or multi_problem() object.
solution	base::data.frame() or tibble::tibble() 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, though 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 phylogenetic branch is colored according to probability that it is expected to persist into the future (per Faith 2008), based on the projects selected for funding by the solution. 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::ggtree() object, or a tidytree::treedata() object if return_data is TRUE.

Dependencies

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 commands to install it:

```
if (!require(remotes)) install.packages("remotes")
remotes::install_bioc("ggtree")
```

If the installation process fails, please consult the package's online documentation.

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)
```

`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, decisions 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,
  baseline_project_name = NULL
)
```

Arguments

projects	base::data.frame() or tibble::tibble() table containing project data. Here, each row should correspond to a different project and columns should contain data that correspond to each project. It 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 outcome (e.g., probability of species' persistence, amount of land covered by an ecosystem) that would be expected for each feature assuming that the project if successfully completed (using a column for each feature), and (iv) and which actions are associated with which projects (using a column for each action). Additionally, projects must have a baseline project (with a zero cost value) that represents the outcome that would be expected if no other project is successfully completed for each feature. Since each feature is assigned the greatest outcome value based on the projects selected for funding in a solution, the combined benefits of multiple projects can be specified 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	base::data.frame() or tibble::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_action_constraints() and add_locked_out_action_constraints()). It should also contain a zero-cost baseline action that is associated with the baseline project.
features	base::data.frame() or tibble::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 expected outcome associated with a particular project be adjusted to account for the outcome associated with the baseline project in the event that the particular project is not successfully competed? This should almost always be set to TRUE to ensure that the calculations do not assume that failing to successfully complete a funded project will result in an outcome with a value of zero (which is often worse than the baseline project). Defaults to TRUE.

baseline_project_name
: character name of the baseline project. Defaults to NULL such that the baseline project name is automatically inferred based on which action has a zero cost. Note that if multiple actions have zero costs, then baseline_project_name must be specified.

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 unction 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 new ProjectProblem 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_wtd_sum_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)
```

`problem_names` *Problem names*

Description

Get the names of the problems in an object.

Usage

```
problem_names(x)

## S4 method for signature 'MultiObjProjectProblem'
problem_names(x)
```

Arguments

x	`multi_problem()` object.

Value

A `character` vector.

Examples

```
# load data
data(sim_multi_projects)
data(sim_multi_features)
data(sim_multi_actions)
data(sim_multi_tree)

# build problem
p <-
  multi_problem(
    obj1 =
      problem(
        sim_multi_projects[[1]], sim_multi_actions, sim_multi_features[[1]],
```

```
      "name", "success", "name", "cost", "name",
      baseline_project_name = "baseline_project_obj1"
    ) %>%
    add_max_phylo_div_objective(
      budget = 200, tree = sim_multi_tree[[1]]
    ) %>%
    add_binary_decisions(),
  obj2 =
    problem(
      sim_multi_projects[[2]], sim_multi_actions, sim_multi_features[[2]],
      "name", "success", "name", "cost", "name",
      baseline_project_name = "baseline_project_obj2"
    ) %>%
    add_max_richness_objective(budget = 200) %>%
    add_binary_decisions(),
  obj3 =
    problem(
      sim_multi_projects[[3]], sim_multi_actions, sim_multi_features[[3]],
      "name", "success", "name", "cost", "name",
      baseline_project_name = "baseline_project_obj3"
    ) %>%
    add_max_wtd_sum_objective(budget = 200) %>%
    add_binary_decisions()
)

# print problem
print(p)

# print problem names
problem_names(p)
```

`ProjectModifier-class` *Conservation problem modifier class*

Description

This super-class is used to represent prototypes that in turn are used to modify a ProjectProblem object. Specifically, the Constraint, Decision, Objective, and Target prototypes inherit from this class. **Only experts should use the fields and methods for this class directly.**

Public fields

`name` `character` value.

`data` `list` containing data.

`internal` `list` containing internal computed values.

Methods

Public methods:

- ProjectModifier$print()
- ProjectModifier$show()
- ProjectModifier$repr()
- ProjectModifier$get_data()
- ProjectModifier$set_data()
- ProjectModifier$get_internal()
- ProjectModifier$set_internal()
- ProjectModifier$calculate()
- ProjectModifier$apply()
- ProjectModifier$clone()

`ProjectModifier$print()`: Print information about the object.

Usage:

```
ProjectModifier$print()
```

Returns: None.

`ProjectModifier$show()`: Print information about the object.

Usage:

```
ProjectModifier$show()
```

Returns: None.

`ProjectModifier$repr()`: Generate a character representation of the object.

Usage:

```
ProjectModifier$repr()
```

Returns: A `character` value.

`ProjectModifier$get_data()`: Get values stored in the `data` field.

Usage:

```
ProjectModifier$get_data(x)
```

Arguments:

`x` `character` name of data.

Returns: An object. If the `data` field does not contain an object associated with the argument to `x`, then a `new_waiver()` object is returned. Set values stored in the `data` field. Note that this method will overwrite existing data.

`ProjectModifier$set_data()`:

Usage:

```
ProjectModifier$set_data(x, value)
```

Arguments:

`x` `character` name of data.

value Object to store.

Returns: Invisible TRUE.

`ProjectModifier$get_internal()`: Get values stored in the `internal` field.

Usage:

```
ProjectModifier$get_internal(x)
```

Arguments:

x `character` name of data.

Returns: An object. If the `internal` field does not contain an object associated with the argument to x, then a `new_waiver()` object is returned.

`ProjectModifier$set_internal()`: Set values stored in the `internal` field. Note that this method will overwrite existing data.

Usage:

```
ProjectModifier$set_internal(x, value)
```

Arguments:

x `character` name of data.

value Object to store.

Returns: An object. If the `internal` field does not contain an object associated with the argument to x, then a `new_waiver()` object is returned.

`ProjectModifier$calculate()`: Perform computations that need to be completed before applying the object.

Usage:

```
ProjectModifier$calculate(x, y)
```

Arguments:

x `new_optimization_problem()` object.

y `problem()` object.

Returns: Invisible TRUE.

`ProjectModifier$apply()`: Update an optimization problem formulation.

Usage:

```
ProjectModifier$apply(x)
```

Arguments:

x `new_optimization_problem()` object.

Returns: Invisible TRUE.

`ProjectModifier$clone()`: The objects of this class are cloneable with this method.

Usage:

```
ProjectModifier$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other classes: Constraint-class, Decision-class, MultiObjApproach-class, MultiObjProjectProblem-class, Objective-class, OptimizationProblem-class, ProjectProblem-class, Solver-class, Target-class, Weight-class

---

`ProjectProblem-class` *Project problem class*

Description

Project problem class

Description

This class is used to represent project prioritization problems. **Only experts should use the fields and methods for this class directly.**

Public fields

`data` `list` containing data (e.g., projects, features).

`defaults` `list` indicating if other fields contain defaults.

`objective` Objective object used to specify the objective function.

`weights` Weight object used to specify the objective function.

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

`targets` Target object used to represent representation targets for features.

`constraints` `list` object used to store Constraint objects for the problem.

`solver` Solver object used to specify the process for generating a solution.

Methods

Public methods:

- `ProjectProblem$new()`
- `ProjectProblem$print()`
- `ProjectProblem$show()`
- `ProjectProblem$repr()`
- `ProjectProblem$get_data()`
- `ProjectProblem$set_data()`
- `ProjectProblem$number_of_actions()`
- `ProjectProblem$number_of_projects()`
- `ProjectProblem$number_of_features()`
- `ProjectProblem$action_names()`
- `ProjectProblem$project_names()`
- `ProjectProblem$feature_names()`

- ProjectProblem$feature_weights()
- ProjectProblem$feature_targets()
- ProjectProblem$feature_phylogeny()
- ProjectProblem$action_costs()
- ProjectProblem$project_costs()
- ProjectProblem$project_success_probabilities()
- ProjectProblem$of_matrix()
- ProjectProblem$pa_matrix()
- ProjectProblem$eof_matrix()
- ProjectProblem$add_solver()
- ProjectProblem$add_targets()
- ProjectProblem$add_weights()
- ProjectProblem$add_objective()
- ProjectProblem$add_decisions()
- ProjectProblem$add_constraint()
- ProjectProblem$clone()

`ProjectProblem$new()`: Create a new conservation problem object.

Usage:

```
ProjectProblem$new(data = list())
```

Arguments:

`data` `list` containing data

Returns: A new `ProjectProblem` object.

`ProjectProblem$print()`: Print concise information about the object.

Usage:

```
ProjectProblem$print()
```

Returns: Invisible `TRUE`.

`ProjectProblem$show()`: Display concise information about the object.

Usage:

```
ProjectProblem$show()
```

Returns: Invisible `TRUE`.

`ProjectProblem$repr()`: Generate a character representation of the object.

Usage:

```
ProjectProblem$repr()
```

Returns: A `character` value.

`ProjectProblem$get_data()`: Get values stored in the `data` field.

Usage:

```
ProjectProblem$get_data(x)
```

Arguments:

`x` `character` name of data.

Returns: An object. If the `data` field does not contain an object associated with the argument to `x`, then a `new_waiver()` object is returned.

`ProjectProblem$set_data()`: Set values stored in the `data` field. Note that this method will overwrite existing data.

Usage:

```
ProjectProblem$set_data(x, value)
```

Arguments:

`x` `character` name of data.

`value` Object to store.

Returns: Invisible `TRUE`.

`ProjectProblem$number_of_actions()`: Obtain the number of actions.

Usage:

```
ProjectProblem$number_of_actions()
```

Returns: An `integer` value.

`ProjectProblem$number_of_projects()`: Obtain the number of projects.

Usage:

```
ProjectProblem$number_of_projects()
```

Returns: An `integer` value.

`ProjectProblem$number_of_features()`: Obtain the number of features.

Usage:

```
ProjectProblem$number_of_features()
```

Returns: An `integer` value.

`ProjectProblem$action_names()`: Obtain the names of the actions.

Usage:

```
ProjectProblem$action_names()
```

Returns: A `character` vector.

`ProjectProblem$project_names()`: Obtain the names of the projects.

Usage:

```
ProjectProblem$project_names()
```

Returns: A `character` vector.

`ProjectProblem$feature_names()`: Obtain the names of the features.

Usage:

```
ProjectProblem$feature_names()
```

Returns: A `character` vector.

`ProjectProblem$feature_weights()`: Obtain the feature weights.

Usage:

```
ProjectProblem$feature_weights()
```

Returns: A named `numeric` vector.

`ProjectProblem$feature_targets()`: Obtain the feature targets.

Usage:

```
ProjectProblem$feature_targets()
```

Returns: A `tibble::tibble()` object.

`ProjectProblem$feature_phylogeny()`: Obtain the feature phylogeny.

Usage:

```
ProjectProblem$feature_phylogeny()
```

Returns: A `ape::phylo()` phylogenetic tree object.

`ProjectProblem$action_costs()`: Obtain the action costs.

Usage:

```
ProjectProblem$action_costs()
```

Returns: A `numeric` vector.

`ProjectProblem$project_costs()`: Obtain the project costs.

Usage:

```
ProjectProblem$project_costs()
```

Returns: A `numeric` vector.

`ProjectProblem$project_success_probabilities()`: Obtain the probability that each project will succeed if funded.

Usage:

```
ProjectProblem$project_success_probabilities()
```

Returns: A `numeric` vector.

`ProjectProblem$of_matrix()`: Obtain information on the outcome for each feature that would be expected if each project funded and is successfully completed.

Usage:

```
ProjectProblem$of_matrix()
```

Returns: A Matrix::dgCMatrix object.

`ProjectProblem$pa_matrix()`: Obtain information on which actions are associated with each project.

Usage:

```
ProjectProblem$pa_matrix()
```

Returns: A Matrix::dgCMatrix object.

`ProjectProblem$eof_matrix()`: Calculate the expected outcome for each feature assuming that each project is funded and accounting for the possibility that funded projects may fail to be successfully completed.

Usage:

```
ProjectProblem$eof_matrix()
```

Returns: A Matrix::dgCMatrix object.

`ProjectProblem$add_solver()`: Create a new object with a solver added to the problem formulation.

Usage:

```
ProjectProblem$add_solver(x)
```

Arguments:

`x` Solver object.

Returns: An updated `ProjectProblem` object.

`ProjectProblem$add_targets()`: Create a new object with targets added to the problem formulation.

Usage:

```
ProjectProblem$add_targets(x)
```

Arguments:

`x` Target object.

Returns: An updated `ProjectProblem` object.

`ProjectProblem$add_weights()`: Create a new object with weights added to the problem formulation.

Usage:

```
ProjectProblem$add_weights(x)
```

Arguments:

`x` Weight object.

Returns: An updated `ProjectProblem` object.

`ProjectProblem$add_objective()`: Create a new object with the objective added to the problem formulation.

Usage:

```
ProjectProblem$add_objective(x)
```

Arguments:

`x` Objective object.

Returns: An updated `ProjectProblem` object.

`ProjectProblem$add_decisions()`: Create a new object with the decisions added to the problem formulation.

Usage:

```
ProjectProblem$add_decisions(x)
```

Arguments:

x Objective object.

Returns: An updated `ProjectProblem` object.

`ProjectProblem$add_constraint()`: Create a new object with the constraint added to the problem formulation.

Usage:

```
ProjectProblem$add_constraint(x)
```

Arguments:

x Constraint object.

Returns: An updated `ProjectProblem` object.

`ProjectProblem$clone()`: The objects of this class are cloneable with this method.

Usage:

```
ProjectProblem$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other classes: `Constraint-class`, `Decision-class`, `MultiObjApproach-class`, `MultiObjProjectProblem-class`, `Objective-class`, `OptimizationProblem-class`, `ProjectModifier-class`, `Solver-class`, `Target-class`, `Weight-class`

---

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	`problem()` object.

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::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

Other functions for evaluating solutions: `rank_importance()`, `replacement_costs()`, `solution_statistics()`

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_wtd_sum_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

Get the names of the projects in an object.

Usage

```
project_names(x)

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

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

Arguments

x	`problem()` or `multi_problem()` object.

Value

A `character` vector or `list` of `character` vectors.

Examples

```
# 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_wtd_sum_objective(budget = 200) %>%
  add_binary_decisions() %>%
  add_default_solver()

# print problem
print(p)

# print project names
project_names(p)
```

`rank_importance` *Rank importance*

Description

Calculate the rank importance for projects in a project prioritization `problem()` (Jung *et al.* 2021). Projects associated with a higher rank value are more irreplaceable, and may need to be implemented sooner than those with lower values.

Usage

```
rank_importance(x, solution, n = 1, ranks = 10, budgets = NULL, ...)
```

Arguments

`x`	`problem()` or `multi_problem()` object.
`solution`	`base::data.frame()` or `tibble::tibble()` 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, though 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.
`ranks`	`integer` number to incremental ranks to evaluate importance. Defaults to 10.

budgets — numeric budget values for generating prioritizations at each increment. This parameter can be used instead of ranks to specify the number of incremental ranks and also the budget values that should be considered for each rank. Defaults to NULL.

... — Arguments passed to solve().

Details

This method involves generating a series of incremental prioritizations, that start with relatively few projects selected and then iteratively selecting additional projects. Projects that are selected at the first increment are assigned the highest importance score and those selected in subsequent increments are assigned lower importance score. Missing (NA) values are assigned to projects which are not selected for funding in the specified solution.

Value

A tibble::tibble() table containing the following columns.

"project" character name of each project.

"rank" integer rank where the project was first selected.

"score" numeric importance score.

References

Jung M, Arnell A, de Lamo X, et al. (2021) Areas of global importance for conserving terrestrial biodiversity, carbon and water. *Nature Ecology and Evolution*, **5**, 1499–1509.

See Also

Other functions for evaluating solutions: project_cost_effectiveness(), replacement_costs(), solution_statistics()

Examples

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

# build problem with maximum weighted sum objective and $400 budget
p <-
  problem(
    sim_projects, sim_actions, sim_features,
    "name", "success", "name", "cost", "name"
  ) %>%
  add_max_wtd_sum_objective(budget = 400) %>%
  add_feature_weights("weight") %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

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

```
# print solution
print(s)

# calculate rank importance values
r <- rank_importance(p, s)

# print output
print(r)
```

`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 irreplaceable, and may need to be implemented sooner than actions with lower replacement cost values.

Usage

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

Arguments

`x`	`problem()` or `multi_problem()` object.
`solution`	`base::data.frame()` or `tibble::tibble()` 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, though 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::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

Other functions for evaluating solutions: project_cost_effectiveness(), rank_importance(), solution_statistics()

Examples

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

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

---

run_example *Run example?*

Description

Determine if the session is suitable for executing long-running examples.

Usage

```
run_example()
```

Details

This function will return TRUE if the session is interactive. Otherwise, it will only return TRUE if the session does not have system environmental variables that indicate that the session is being used for package checks, or for building documentation.

Value

A logical value.

Examples

```
# should examples be run in current environment?
run_example()
```

---

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 'OptimizationProblem'
show(x)

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

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

Arguments

x	Any object.

Value

None.

See Also

`methods::show()`.

`simulate_multi_ppp_data` *Simulate multi-objective data for the 'Project Prioritization Protocol'*

Description

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

Usage

```
simulate_multi_ppp_data(
  number_objectives,
  number_features,
  number_actions,
  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_objectives	Number of objectives for which to simulate data.
number_features	numeric number of features.
number_actions	Number of actions for which to simulate data.
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 a set of projects, wherein each project is assigned to a particular objective and is associated with a particular feature. Although multiple projects may be assigned to the same objective, note that each project is associated with a different feature. Also note that that each objective is associated with a "baseline" "baseline" (do nothing) project to reflect features'

persistence when their conservation project is not funded. Specifically, the data are simulated using the following procedure.

1. A set of objectives are defined (per `number_objectives`) and a set of features are defined (per `number_features`).
2. Each feature is then randomly assigned to each objective. Note that each objective will always have at least one of feature.
3. A set of actions (per `number_actions`) are simulated and the costs for these actions are simulated using a normal distribution and the `cost_mean` and `cost_sd` arguments. In addition to these actions, a set of additional baseline actions are simulated for each objective.
4. A set of projects are created for each feature by randomly selecting a set of actions.
5. 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.
6. 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.
7. 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.
8. An additional project is created for each programme 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.
9. A phylogenetic tree is simulated for the features that belong to each objective (separately) using `ape::rcoal()`.
10. 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 following elements.

`"projects_obj1"`, **...**, `"projects_objN"` A `list` of `tibble::tibble()` containing data for the conservation projects associated with each objective. Each element contains a data frame with the following 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 probability that each feature will persist if it is funded and successfully completed. 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::tibble() containing the data for the conservation actions across all objectives. 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_obj1", ..., "features_objN" A list of tibble::tibble() containing data for the features (e.g., species) associated with each objective. Each element contains a data frame with the following 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_obj1", ..., "tree_objN" A list of ape::phylo() phylogenetic tree objects for the features associated with each objective (separately).

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_multi_ppp_data(
  number_objectives = 3,
  number_features = 7,
  number_actions = 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_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 using the following procedure.

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 `ape::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 following elements.

`"projects"` A `tibble::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 probability that each feature will persist if it is funded and successfully completed. 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::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::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" An ape::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 using the following procedure.

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 `ape::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 following elements.

`"projects"` A `tibble::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 probability that each feature will persist if it is funded and successfully completed. 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::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::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" An `ape::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*, **56**: 481–490.

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 conservation projects based on a single objective.

Usage

```
data(sim_actions)

data(sim_projects)

data(sim_features)

data(sim_tree)
```

Format

sim_projects `tibble::tibble()` object.

sim_actions `tibble::tibble()` object.

sim_features `tibble::tibble()` object.

sim_tree `ape::phylo()` object.

Details

The data set contains the following objects:

`sim_projects` A `tibble::tibble()` object 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"` This column contains the `character` name for each project.
- `"success"` This column contains `numeric` values that denote probability of each project being successfully completed if it is funded.
- `"F1"` **...** `"F5"` These columns (i.e., `"F1"`, `"F2"`, `"F3"`, `"F4"`, `"F5"`) contain `numeric` values that indicate the probability that each feature will persist if the project is successfully completed. Missing values (`NA`) indicate that a feature does not benefit from a project being funded.
- `"F1_action"` **...** `"F5_action"` These columns (i.e., `"F1_action"`, `"F2_action"`, `"F3_action"`, `"F4_action"`, `"F5_action"`) contain `logical` (`TRUE`/`FALSE`) values that indicate if each action is associated with each project or not.
- `"baseline_action"` This column contains `logical` values indicating if each project is the baseline project or not.

`sim_actions` A `tibble::tibble()` object 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"` This column contains the `character` name for each action.
- `"cost"` This column contains `numeric` values denoting the cost for each action.
- `"locked_in"` This column contains `logical` values indicating if particular actions should be locked into the solution.
- `"locked_out"` This column contains `logical` values indicating if particular actions should be locked out of the solution.

`sim_features` A `tibble::tibble()` object 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"` This column contains the `character` name for each feature.
- `"weight"` This column contains `numeric` values denoting the weight for each feature.

tree A `ape::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)
```

`sim_multi_data` *Simulated multi-objective data*

Description

Simulated data for prioritizing conservation projects based on multiple objectives.

Usage

```
data(sim_multi_actions)

data(sim_multi_projects)

data(sim_multi_features)

data(sim_multi_tree)
```

Format

sim_multi_projects `list` of `tibble::tibble()` objects.

sim_multi_actions `tibble::tibble()` object.

sim_multi_features `list` of `tibble::tibble()` objects.

sim_multi_tree `list` of `ape::phylo()` objects.

Details

The data set contains the following objects.

`"sim_multi_projects"` A `list` of three `tibble::tibble()` objects containing data for simulated conservation projects. Each `tibble::tibble()` object corresponds to a different objective. Within each of these `tibble::tibble()` objects, each row corresponds to a different project and each column contains information about the projects. These tables contains the following columns.

- `"name"` These columns contain the `character` name for each project.
- `"success"` These columns contain the `numeric` probability of each project succeeding if it is funded.
- `"F1"` **...** `"F5"` These columns contain `numeric` values for each feature (i.e., `"F1"`, `"F2"`, `"F3"`, `"F4"`, ..., `"F10"`) that indicate the probability that the feature will persist if each project is successfully completed. Missing values (`NA`) indicate that a feature does not benefit from a project being funded.

"A1" ... "A15" These columns contain logical (TRUE/FALSE) values for each action (i.e., "A1", "A2", ..., "A15") indicating if the action is associated with each project or not.

"baseline_action" These columns contain logical (TRUE/FALSE) values for each project indicating if the project is the baseline project or not.

sim_multi_actions A tibble::tibble() object containing data for 15 simulated actions. Each row corresponds to a different action and each column contains information about the actions. This table contains the following columns.

"name" This column contains the character name for each action.

"cost" This column contains the numeric cost for each action.

"locked_in" This column contains logical values indicating if particular actions should be locked into the solution.

"locked_out" This column contains logical values indicating if particular actions should be locked out of the solution.

sim_multi_features A list of three tibble::tibble() objects containing data for ten simulated features. Each tibble::tibble() object corresponds to a different objective. With each tibble::tibble() object, each row corresponds to a different feature and each column contains information about the features. These tables contains the following columns.

"name" These columns contain character values denoting name for each feature.

"weight" These columns contain numeric values denoting weight for each feature.

sim_multi_trees A list of ape::phylo() phylogenetic tree objects for the features associated with each of the three objective.

Examples

```
# load data
data(sim_multi_projects)
data(sim_multi_actions)
data(sim_multi_features)
data(sim_multi_tree)

# print project data
print(sim_multi_projects)

# print action data
print(sim_multi_actions)

# print feature data
print(sim_multi_features)

# print phylogenetic trees
print(sim_multi_tree)
```

solution_statistics *Solution statistics*

Description

Calculate statistics to describe a solution to a project prioritization problem.

Usage

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

Arguments

x	problem() or multi_problem() object.
solution	base::data.frame() or tibble::tibble() 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, though they will be ignored.

Value

A tibble::tibble() containing the following columns.

"cost" This column contains numeric values describing the cost of each solution.

"obj" This column contains numeric values describing the objective value for each solution. This is calculated using the objective function defined for the argument to x. Note that if x is a multi_problem() object, then an objective column will be created for each problem in x.

x$project_names() These columns contain logical values that indicate if each project had all of its actions selected for funding or not.

x$feature_names() These columns contain numeric values that describe the expected outcome for each feature based on the actions selected for funding.

See Also

Other functions for evaluating solutions: project_cost_effectiveness(), rank_importance(), 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_wtd_sum_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 for the solutions
solution_statistics(p, solutions)
```

`solve` *Solve*

Description

Solve a conservation planning `problem()`.

Usage

```
## S4 method for signature 'OptimizationProblem,Solver'
solve(a, b, ...)
```

```
## S4 method for signature 'ProjectProblem,missing'
solve(a, b, ...)

## S4 method for signature 'MultiObjProjectProblem,missing'
solve(a, b, ...)
```

Arguments

a	problem(), multi_problem(), or OptimizationProblem object.
b	Solver object. Note that this parameter is only used if a is an OptimizationProblem 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 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 a problem() or multi_problem() object, then a tibble::tibble() 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. In particular, it will have the following columns.

"solution" This column contains integer identifiers for the solutions.

"status" This column contains character values that describe the solver status. For example, these values may indicate if the solver returned an optimal or suboptimal solution.

"obj" This column contains numeric values that contain the objective value for each solution. This is calculated using the objective function defined for the argument to a. Note that if a is a multi_problem() object, then an objective column will be created for each problem in a.

"cost" This column contains numeric values that describe the total cost associated with each solution.

x$action_names() These columns contain logical (TRUE/FALSE) values that indicate if each for each action was selected for funding (or not) by each solution.

x$project_names() These columns contain logical (TRUE/FALSE) values that indicate if each for each project had all of its actions selected for funding (or not) by each solution.

x$feature_names() These columns containnumeric‘ values that describe the expected outcome for each feature based on the actions selected for funding.

See Also

The solution_statistics() function can be used to compute these statistics for solutions. This may be useful to evaluate the performance of solutions generated based on expert opinion, or solutions according to objectives that are different from those used to generate them.

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_wtd_sum_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 class*

Description

This class is used to represent solvers for optimization. **Only experts should use the fields and methods for this class directly.**

Super class

`ProjectModifier` -> `Solver`

Public fields

`has_pwlobj` `logical` indicating if solver supports piece-wise linear components in an objective function.

Methods

Public methods:

- `Solver$set_start_solution()`
- `Solver$remove_start_solution()`
- `Solver$solve()`
- `Solver$clone()`

`Solver$set_start_solution()`: Set start solution.

Usage:

```
Solver$set_start_solution(x)
```

Arguments:

`x` `numeric` vector.

Returns: Invisible `TRUE`.

`Solver$remove_start_solution()`: Remove start solution.

Usage:

```
Solver$remove_start_solution(x)
```

Arguments:

`x` `numeric` vector.

Returns: Invisible `TRUE`.

`Solver$solve()`: Solve an optimization problem.

Usage:

```
Solver$solve(x, ...)
```

Arguments:

`x` `new_optimization_problem()` object.

`...` Additional arguments as needed.

Returns: Invisible `TRUE`.

`Solver$clone()`: The objects of this class are cloneable with this method.

Usage:

```
Solver$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other classes: Constraint-class, Decision-class, MultiObjApproach-class, MultiObjProjectProblem-class, Objective-class, OptimizationProblem-class, ProjectModifier-class, ProjectProblem-class, Target-class, Weight-class

---

solvers *Solvers*

Description

Solvers specify the software and configuration used to generate solutions for a project prioritization problem. By default, the best available exact algorithm solver is used.

Details

The following solvers can be used to generate solutions for a project prioritization problem.

add_default_solver() Add the best installed solver.

add_gurobi_solver() Add a solver to generate solutions with the *Gurobi* software.

add_highs_solver() Add a solver to generate solutions with the *HiGHS* software via the **highs** package.

add_cbc_solver() Add a solver to generate solutions with the *CBC* software via the **rcbc** package.

add_rsymphony_solver() Add a solver to generate solutions with the *SYMPHONY* software via the **Rsymphony** package.

add_lpsymphony_solver() Add a solver to generate solutions with the *SYMPHONY* software via the **lpsymphony** package.

add_lpsolveapi_solver() Add a solver to generate solutions with the *lp_solve* software via the **lpSolveAPI** package.

add_heuristic_solver() Add a solver to generate solutions using a backwards heuristic algorithm.

add_random_solver() Add a solver to generate solutions by randomly selecting actions for funding.

See Also

Other overviews: approaches, constraints, objectives, targets, weights

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_wtd_sum_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 highs solver
p4 <- p1 %>% add_highs_solver()

# build another problem, with the cbc solver
p5 <- p1 %>% add_cbc_solver()

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

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

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

# build another problem, with the heuristic solver
p9 <- p1 %>% add_heuristic_solver()

# build another problem, with the random solver
p10 <- 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), solve(p9), solve(p10)
)
s$solver <- c(
  "default", "gurobi", "highs", "cbc", "Rsymphony", "lpsymphony",
  "lpSolveAPI", "heuristic", "random"
)

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

Target-class *Target class*

Description

This class is used to represent targets for optimization. **Only experts should use the fields and methods for this class directly.**

Super class

`ProjectModifier` -> `Target`

Methods

Public methods:

- `Target$output()`
- `Target$clone()`

`Target$output()`: Output the targets.

Usage:

```
Target$output()
```

Returns: `tibble::tibble()` data frame.

`Target$clone()`: The objects of this class are cloneable with this method.

Usage:

```
Target$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other classes: `Constraint-class`, `Decision-class`, `MultiObjApproach-class`, `MultiObjProjectProblem-class`, `Objective-class`, `OptimizationProblem-class`, `ProjectModifier-class`, `ProjectProblem-class`, `Solver-class`, `Weight-class`

`targets` *Targets*

Description

Targets are used to specify a threshold minimum outcome for each feature in a project prioritization problem. 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

Other overviews: `approaches`, `constraints`, `objectives`, `solvers`, `weights`

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::tibble()` objects.

Usage

```
nrow(x)

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

ncol(x)

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

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

Arguments

x	tibble::tibble() object.

Details

The following methods are provided from manipulating tibble::tibble() objects.

nrow integer number of rows.

ncol 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 class*

Description

This class is used to represent targets for optimization. **Only experts should use the fields and methods for this class directly.**

Super class

ProjectModifier -> Weight

Methods

Public methods:

- Weight$output()
- Weight$clone()

`Weight$output()`: Output the targets.

Usage:

```
Weight$output()
```

Returns: A `numeric` matrix.

`Weight$clone()`: The objects of this class are cloneable with this method.

Usage:

```
Weight$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other classes: `Constraint-class`, `Decision-class`, `MultiObjApproach-class`, `MultiObjProjectProblem-class`, `Objective-class`, `OptimizationProblem-class`, `ProjectModifier-class`, `ProjectProblem-class`, `Solver-class`, `Target-class`

`weights` *Weights*

Description

Weights are used to specify the relative importance for particular features in a project prioritization problem. 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

The following functions can be used to add weights to a project prioritization problem.

`add_default_weights()` Add default weights.

`add_feature_weights()` Add different weights for each feature.

See Also

Other overviews: `approaches`, `constraints`, `objectives`, `solvers`, `targets`

Examples

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

# build problem with maximum weighted sum objective and $300 budget
p1 <-
  problem(
    sim_projects, sim_actions, sim_features,
```

```
    "name", "success", "name", "cost", "name"
  ) %>%
  add_max_wtd_sum_objective(budget = 200) %>%
  add_binary_decisions()

# build problem with default weights
p2 <- p1 %>% add_default_weights()

# build problem with feature weights
p3 <- p1 %>% add_feature_weights("weight")

# generate solutions using
s <- rbind(solve(p2), solve(p3))
s$weights <- c("default", "weights")

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

Index