Package ‘prioritizr’

March 31, 2021

Type Package

Version 7.0.1

Title Systematic Conservation Prioritization in R

Description Systematic conservation prioritization using mixed integer linear programming (MILP). It provides a flexible interface for building and solving conservation planning problems. Once built, conservation planning problems can be solved using a variety of commercial and open-source exact algorithm solvers. By using exact algorithm solvers, solutions can be generated that are guaranteed to be optimal (or within a pre-specified optimality gap). Furthermore, conservation problems can be constructed to optimize the spatial allocation of different management actions or zones, meaning that conservation practitioners can identify solutions that benefit multiple stakeholders. To solve large-scale or complex conservation planning problems, users should install the Gurobi optimization software (available from <https://www.gurobi.com/>) and the 'gurobi' R package (see Gurobi Installation Guide vignette for details). Additionally, the 'rcbc' R package (available at <https://github.com/dirkschumacher/rcbc>) can be used to generate solutions using the CBC optimization software (<https://projects.coin-or.org/Cbc>).

Imports utils, methods, assertthat (>= 0.2.0), data.table, uuid, Matrix, slam (>= 0.1-48), igraph, ape, rgeos, plyr, parallel, doParallel, magrittr, exactextractr (>= 0.2.0), fasterize (>= 1.0.2), tibble (>= 2.0.0), withr (>= 2.3.0)

Suggests testthat, knitr, roxygen2, scales, xtable, RandomFields, maptools, PBSmapping, gurobi, rcbc (>= 0.1.0.9001), cplexAPI (>= 1.4.0), lpsymphony, Rsymphony, rmarkdown, prioritizrdata (>= 0.2.4)

Depends R (>= 3.5.0), raster, sp, sf (>= 0.8-0), proto

LinkingTo Rcpp, RcppArmadillo (>= 0.10.1.0.0), BH

License GPL-3

Language en-US

Encoding UTF-8

LazyData true

SystemRequirements C++11

URL https://prioritizr.net, https://github.com/prioritizr/prioritizr

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

VignetteBuilder knitr

RoxygenNote 7.1.1

Collate 'internal.R' 'pproto.R' 'Parameter-proto.R' 'ArrayParameter-proto.R' 'MiscParameter-proto.R' 'Parameters-proto.R' 'ScalarParameter-proto.R' 'parameters.R' 'waiver.R' 'ConservationModifier-proto.R' 'Penalty-proto.R' 'Constraint-proto.R' 'Collection-proto.R' 'category_vector.R' 'category_layer.R' 'binary_stack.R' 'ConservationProblem-proto.R' 'Decision-proto.R' 'Id.R' 'Objective-proto.R' 'OptimizationProblem-proto.R' 'OptimizationProblem-methods.R' 'Portfolio-proto.R' 'RcppExports.R' 'Solver-proto.R' 'Target-proto.R' 'zones.R' 'add_absolute_targets.R' 'add_binary_decisions.R' 'marxan_boundary_data_to_matrix.R' 'add_boundary_penalties.R' 'add_cbc_solver.R' 'add_connectivity_penalties.R' 'add_contiguity_constraints.R' 'add_cplex_solver.R' 'add_cuts_portfolio.R' 'add_default_decisions.R' 'add_default_objective.R' 'add_default_portfolio.R' 'add_default_solver.R' 'add_default_targets.R' 'add_extra_portfolio.R' 'add_feature_contiguity_constraints.R' 'add_feature_weights.R' 'add_gap_portfolio.R' 'add_gurobi_solver.R' 'add_linear_penalties.R' 'intersecting_units.R' 'add_locked_in_constraints.R' 'add_locked_out_constraints.R' 'loglinear_interpolation.R' 'add_loglinear_targets.R' 'add_lpsymphony_solver.R' 'add_mandatory_allocation_constraints.R' 'tbl_df.R' 'add_manual_targets.R' 'add_manual_bounded_constraints.R' 'add_manual_locked_constraints.R' 'add_max_cover_objective.R' 'add_max_features_objective.R' 'add_max_phylo_div_objective.R' 'add_max_phylo_end_objective.R' 'add_max_utility_objective.R' 'add_min_largest_shortfall_objective.R' 'add_min_set_objective.R' 'add_min_shortfall_objective.R' 'add_neighbor_contraints.R' 'add_proportion_decisions.R' 'add_relative_targets.R' 'add_rsymphony_solver.R' 'add_semicontinuous_decisions.R' 'add_shuffle_portfolio.R' 'add_top_portfolio.R' 'adjacency_matrix.R' 'boundary_matrix.R' 'branch_matrix.R' 'compile.R' 'connectivity_matrix.R' 'constraints.R' 'data.R' 'decisions.R' 'deprecated.R' 'distribute_load.R' 'eval_boundary_summary.R' 'eval_connectivity_summary.R' 'eval_cost_summary.R' 'eval_feature_representation_summary.R' 'eval_ferrier_importance.R' 'eval_n_summary.R' 'problem.R' 'eval_rare_richness_importance.R' 'presolve_check.R' 'solve.R'

'eval_replacement_importance.R'
'eval_target_coverage_summary.R' 'fast_extract.R'
'feature_abundances.R' 'feature_names.R' 'importance.R'
'magrittr-operators.R' 'marxan_problem.R' 'misc.R'
'new_optimization_problem.R' 'number_of_features.R'
'number_of_planning_units.R' 'number_of_total_units.R'
'number_of_zones.R' 'objectives.R' 'package.R' 'penalties.R'
'planning_unit_solution_status.R' 'portfolios.R'
'predefined_optimization_problem.R' 'print.R'
'proximity_matrix.R' 'rij_matrix.R' 'run_calculations.R' 'sf.R'
'show.R' 'simulate.R' 'solvers.R' 'summaries.R' 'targets.R'
'write_problem.R' 'zone_names.R' 'zzz.R'

NeedsCompilation yes

Author Jeffrey O Hanson [aut] (<https://orcid.org/0000-0002-4716-6134>),
Richard Schuster [aut, cre] (<https://orcid.org/0000-0003-3191-7869>),
Nina Morrell [aut],
Matthew Strimas-Mackey [aut] (<https://orcid.org/0000-0001-8929-7776>),
Matthew E Watts [aut],
Peter Arcese [aut] (<https://orcid.org/0000-0002-8097-482X>),
Joseph Bennett [aut] (<https://orcid.org/0000-0002-3901-9513>),
Hugh P Possingham [aut] (<https://orcid.org/0000-0001-7755-996X>)

Maintainer Richard Schuster <richard.schuster@glel.carleton.ca>

Repository CRAN

Date/Publication 2021-03-31 12:40:02 UTC

R topics documented:

`add_absolute_targets` *Add absolute targets*

Description

Set targets expressed as the actual value of features in the study area that need to be represented in the prioritization. For instance, setting a target of 10 requires that the solution secure a set of planning units for which their summed feature values are equal to or greater than 10.

Usage

```
add_absolute_targets(x, targets)

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

## S4 method for signature 'ConservationProblem,matrix'
add_absolute_targets(x, targets)

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

Arguments

`x`	`problem()` (i.e. `ConservationProblem`) object.
`targets`	Object that specifies the targets for each feature. See the Targets data format section for more information.

Details

Targets are used to specify the minimum amount or proportion of a feature's distribution that needs to be protected. Most conservation planning problems require targets with the exception of the maximum cover (see `add_max_cover_objective()`) and maximum utility (see `add_max_utility_objective()`) problems. Attempting to solve problems with objectives that require targets without specifying targets will throw an error.

For problems associated with multiple management zones, this function can be used to set targets that each pertain to a single feature and a single zone. To set targets which can be met through allocating different planning units to multiple zones, see the `add_manual_targets()` function. An example of a target that could be met through allocations to multiple zones might be where each management zone is expected to result in a different amount of a feature and the target requires that the total amount of the feature in all zones must exceed a certain threshold. In other words, the target does not require that any single zone secure a specific amount of the feature, but the total amount held in all zones must secure a specific amount. Thus the target could, potentially, be met through allocating all planning units to any specific management zone, or through allocating the planning units to different combinations of management zones.

Value

Object (i.e. `ConservationProblem`) with the targets added to it.

Targets data format

The targets for a problem can be specified using the following formats.

`numeric vector` of target values for each feature. Additionally, for convenience, this type of argument can be a single value to assign the same target to each feature. Note that this type of argument cannot be used to specify targets for problems with multiple zones.

`matrix` containing a target for each feature in each zone. Here, each row corresponds to a different feature in argument to `x`, each column corresponds to a different zone in argument to `x`, and each cell contains the target value for a given feature that the solution needs to secure in a given zone.

`character` containing the names of fields (columns) in the feature data associated with the argument to `x` that contain targets. This type of argument can only be used when the feature data associated with `x` is a `data.frame`. This argument must contain a field (column) name for each zone.

See Also

targets.

Examples

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

# load data
data(sim_pu_raster, sim_features, sim_pu_zones_stack, sim_features_zones)
```

```
# create simple problem
p <- problem(sim_pu_raster, sim_features) %>%
     add_min_set_objective() %>%
     add_binary_decisions() %>%
     add_default_solver(verbose = FALSE)

# create problem with targets to secure 3 amounts for each feature
p1 <- p %>% add_absolute_targets(3)

# create problem with varying targets for each feature
targets <- c(1, 2, 3, 2, 1)
p2 <- p %>% add_absolute_targets(targets)
## Not run:
# solve problem
s <- stack(solve(p1), solve(p2))

# plot solution
plot(s, main = c("equal targets", "varying targets"), axes = FALSE,
     box = FALSE)

## End(Not run)

# create a problem with multiple management zones
p3 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create a problem with targets that specify an equal amount of each feature
# to be represented in each zone
p4_targets <- matrix(2, nrow = number_of_features(sim_features_zones),
                     ncol = number_of_zones(sim_features_zones),
                     dimnames = list(feature_names(sim_features_zones),
                                     zone_names(sim_features_zones)))
print(p4_targets)

p4 <- p3 %>% add_absolute_targets(p4_targets)

# solve problem
## Not run:
# solve problem
s4 <- solve(p4)

# plot solution (pixel values correspond to zone identifiers)
plot(category_layer(s4), main = c("equal targets"))

## End(Not run)
# create a problem with targets that require a varying amount of each
# feature to be represented in each zone
p5_targets <- matrix(rpois(15, 1),
                     nrow = number_of_features(sim_features_zones),
                     ncol = number_of_zones(sim_features_zones),
                     dimnames = list(feature_names(sim_features_zones),
```

```
                                     zone_names(sim_features_zones)))
print(p5_targets)

p5 <- p3 %>% add_absolute_targets(p4_targets)
# solve problem
## Not run:
# solve problem
s5 <- solve(p5)

# plot solution (pixel values correspond to zone identifiers)
plot(category_layer(s5), main = c("varying targets"))

## End(Not run)
```

---

add_binary_decisions *Add binary decisions*

Description

Add a binary decision to a conservation planning `problem()`. This is the classic decision of either prioritizing or not prioritizing a planning unit. Typically, this decision has the assumed action of buying the planning unit to include in a protected area network. If no decision is added to a problem then this decision class will be used by default.

Usage

```
add_binary_decisions(x)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object.

Details

Conservation planning problems involve making decisions on planning units. These decisions are then associated with actions (e.g. turning a planning unit into a protected area). If no decision is explicitly added to a problem, then the binary decision class will be used by default. Only a single decision should be added to a `ConservationProblem` object. Note that if multiple decisions are added to a problem object, then the last one to be added will be used.

Value

Object (i.e. `ConservationProblem`) with the decisions added to it.

See Also

decisions.

Examples

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

# load data
data(sim_pu_raster, sim_features, sim_pu_zones_stack, sim_features_zones)

# create minimal problem with binary decisions
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s1 <- solve(p1)

# plot solution
plot(s1, main = "solution")

## End(Not run)
# build multi-zone conservation problem with binary decisions
p2 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve the problem
s2 <- solve(p2)

# print solution
print(s2)

# plot solution
plot(category_layer(s2), main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

`add_boundary_penalties` *Add boundary penalties*

Description

Add penalties to a conservation planning `problem()` to favor solutions that spatially clump planning units together based on the overall boundary length (perimeter).

Usage

```
add_boundary_penalties(
  x,
  penalty,
  edge_factor = rep(0.5, number_of_zones(x)),
  zones = diag(number_of_zones(x)),
  data = NULL
)
```

Arguments

x
: `problem()` (i.e. `ConservationProblem`) object.

penalty
: `numeric` penalty that is used to scale the importance of selecting planning units that are spatially clumped together compared to the main problem objective (e.g. solution cost when the argument to `x` has a minimum set objective per `add_min_set_objective()`). Higher `penalty` values prefer solutions with a higher degree of spatial clumping, and smaller `penalty` values prefer solutions with a smaller degree of clumping. Note that negative `penalty` values prefer solutions that are more spread out. This parameter is equivalent to the boundary length modifier (BLM) parameter in *Marxan*.

edge_factor
: `numeric` proportion to scale planning unit edges (borders) that do not have any neighboring planning units. For example, an edge factor of `0.5` is commonly used to avoid overly penalizing planning units along a coastline. Note that this argument must have an element for each zone in the argument to `x`.

zones
: `matrix` or `Matrix` object describing the clumping scheme for different zones. Each row and column corresponds to a different zone in the argument to `x`, and cell values indicate the relative importance of clumping planning units that are allocated to a combination of zones. Cell values along the diagonal of the matrix represent the relative importance of clumping planning units that are allocated to the same zone. Cell values must range between 1 and -1, where negative values favor solutions that spread out planning units. The default argument to `zones` is an identity matrix (i.e. a matrix with ones along the matrix diagonal and zeros elsewhere), so that penalties are incurred when neighboring planning units are not assigned to the same zone. If the cells along the matrix diagonal contain markedly smaller values than those found elsewhere in the matrix, then solutions are preferred that surround planning units with those allocated to different zones (i.e. greater spatial fragmentation).

data
: `NULL`, `data.frame`, `matrix`, or `Matrix` object containing the boundary data. These data describe the total amount of boundary (perimeter) length for each planning unit, and the amount of boundary (perimeter) length shared between different planning units (i.e. planning units that are adjacent to each other). See the Data format section for more information.

Details

This function adds penalties to a conservation planning problem to penalize fragmented solutions. It was is inspired by Ball *et al.* (2009) and Beyer *et al.* (2016). The `penalty` argument is equivalent to the boundary length modifier (BLM) used in *Marxan*. Note that this function can only be used to represent symmetric relationships between planning units. If asymmetric relationships are required, use the `add_connectivity_penalties()` function.

The boundary penalties are calculated using the following equations. Let I represent the set of planning units (indexed by i or j), Z represent the set of management zones (indexed by z or y), and X_{iz} represent the decision variable for planning unit i for in zone z (e.g. with binary values one indicating if planning unit is allocated or not). Also, let p represent the argument to `penalty`, E_z represent the argument to `edge_factor`, B_{ij} represent the matrix argument to `data` (e.g. generated using `boundary_matrix()`), and W_{zz} represent the matrix argument to `zones`.

$$\sum_{i}^{I}\sum_{j}^{I}\sum_{z}^{Z}(\mathit{ifelse}(i == j, E_z, 1) \times p \times W_{zz} B_{ij}) + \sum_{i}^{I}\sum_{j}^{I}\sum_{z}^{Z}\sum_{y}^{Z}(-2 \times p \times X_{iz} \times X_{jy} \times W_{zy} \times B_{ij})$$

Note that when the problem objective is to maximize some measure of benefit and not minimize some measure of cost, the term p is replaced with $-p$.

Value

Object (i.e. `ConservationProblem`) with the penalties added to it.

Data format

The argument to `data` can be specified using the following formats. Note that boundary data must always describe symmetric relationships between planning units.

`NULL` boundary data are automatically calculated using the `boundary_matrix()` function. This argument is the default. Note that the boundary data must be supplied using one of the other formats below if the planning unit data in the argument to `x` do not explicitly contain spatial information (e.g. planning unit data are a `data.frame` or `numeric` class).

`matrix`, `Matrix` where rows and columns represent different planning units and the value of each cell represents the amount of shared boundary length between two different planning units. Cells that occur along the matrix diagonal represent the amount of exposed boundary associated with each planning unit that has no neighbor (e.g. these value might pertain to boundaries along a coastline).

`data.frame` with the columns `"id1"`, `"id2"`, and `"boundary"`. The `"id1"` and `"id2"` columns contain identifiers (indices) for a pair of planning units, and the `"boundary"` column contains the amount of shared boundary length between these two planning units. This format follows the the standard *Marxan* format for boundary data (i.e. per the "bound.dat" file).

References

Ball IR, Possingham HP, and Watts M (2009) *Marxan and relatives: Software for spatial conservation prioritisation* in Spatial conservation prioritisation: Quantitative methods and computational tools. Eds Moilanen A, Wilson KA, and Possingham HP. Oxford University Press, Oxford, UK.

Beyer HL, Dujardin Y, Watts ME, and Possingham HP (2016) Solving conservation planning problems with integer linear programming. *Ecological Modelling*, 228: 14–22.

See Also

penalties.

Examples

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

# load data
data(sim_pu_raster, sim_features, sim_pu_zones_stack, sim_features_zones)

# create minimal problem
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create problem with low boundary penalties
p2 <- p1 %>% add_boundary_penalties(50, 1)

# create problem with high boundary penalties but outer edges receive
# half the penalty as inner edges
p3 <- p1 %>% add_boundary_penalties(500, 0.5)

# create a problem using precomputed boundary data
bmat <- boundary_matrix(sim_pu_raster)
p4 <- p1 %>% add_boundary_penalties(50, 1, data = bmat)

## Not run:
# solve problems
s <- stack(solve(p1), solve(p2), solve(p3), solve(p4))

# plot solutions
plot(s, main = c("basic solution", "small penalties", "high penalties",
                 "precomputed data"), axes = FALSE, box = FALSE)

## End(Not run)
# create minimal problem with multiple zones and limit the run-time for
# solver to 10 seconds so this example doesn't take too long
p5 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(0.2, nrow = 5, ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(time_limit = 10, verbose = FALSE)

# create zone matrix which favors clumping planning units that are
# allocated to the same zone together - note that this is the default
zm6 <- diag(3)
print(zm6)

# create problem with the zone matrix and low penalties
```

```
p6 <- p5 %>% add_boundary_penalties(50, zone = zm6)

# create another problem with the same zone matrix and higher penalties
p7 <- p5 %>% add_boundary_penalties(500, zone = zm6)

# create zone matrix which favors clumping units that are allocated to
# different zones together
zm8 <- matrix(1, ncol = 3, nrow = 3)
diag(zm8) <- 0
print(zm8)

# create problem with the zone matrix
p8 <- p5 %>% add_boundary_penalties(500, zone = zm8)

# create zone matrix which strongly favors clumping units
# that are allocated to the same zone together. It will also prefer
# clumping planning units in zones 1 and 2 together over having
# these planning units with no neighbors in the solution
zm9 <- diag(3)
zm9[upper.tri(zm9)] <- c(0.3, 0, 0)
zm9[lower.tri(zm9)] <- zm9[upper.tri(zm9)]
print(zm9)

# create problem with the zone matrix
p9 <- p5 %>% add_boundary_penalties(500, zone = zm9)

# create zone matrix which favors clumping planning units in zones 1 and 2
# together, and favors planning units in zone 3 being spread out
# (i.e. negative clumping)
zm10 <- diag(3)
zm10[3, 3] <- -1
print(zm10)

# create problem with the zone matrix
p10 <- p5 %>% add_boundary_penalties(500, zone = zm10)

## Not run:
# solve problems
s2 <- stack(category_layer(solve(p5)), category_layer(solve(p6)),
            category_layer(solve(p7)), category_layer(solve(p8)),
            category_layer(solve(p9)), category_layer(solve(p10)))

# plot solutions
plot(s2, main = c("basic solution", "within zone clumping (low)",
                  "within zone clumping (high)", "between zone clumping",
                  "within + between clumping", "negative clumping"),
     axes = FALSE, box = FALSE)

## End(Not run)
```

---

add_cbc_solver   *Add a* CBC *solver*

Description

Specify that the *CBC* (COIN-OR branch and cut) software (Forrest & Lougee-Heimer 2005) should be used to solve a conservation planning `problem()`. 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 = TRUE,
  threads = 1,
  first_feasible = FALSE,
  verbose = TRUE
)
```

Arguments

`x`	`problem()` (i.e. `ConservationProblem`) 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 0.1 (i.e. 10% from optimality).
`time_limit`	`numeric` time limit (seconds) for generating solutions. The solver will return the current best solution when this time limit is exceeded. The default value is the largest integer value (i.e. `.Machine$integer.max`), effectively meaning that solver will keep running until a solution within the optimality gap is found.
`presolve`	`logical` attempt to simplify the problem before solving it? Defaults to `TRUE`.
`threads`	`integer` number of threads to use for the optimization algorithm. The default value is 1.
`first_feasible`	`logical` should the first feasible solution be be returned? If `first_feasible` is set to `TRUE`, the solver will return the first solution it encounters that meets all the constraints, regardless of solution quality. Note that the first feasible solution is not an arbitrary solution, rather it is derived from the relaxed solution, and is therefore often reasonably close to optimality. Defaults to `FALSE`.
`verbose`	`logical` should information be printed while solving optimization problems? 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. Although formal benchmarks examining the performance of this solver for conservation planning problems have yet to be completed, preliminary analyses suggest that it performs much faster than the other open-source solvers (i.e.

add_rsymphony_solver(), add_rsymphony_solver()), and so we recommend using this solver if the *Gurobi* and *IBM CPLEX* solvers are unavailable.

Value

Object (i.e. ConservationProblem) 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 official installation instructions for the package.

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

solvers.

Examples

```
## Not run:
# load data
data(sim_pu_raster, sim_features)

# create problem
p <- problem(sim_pu_raster, sim_features) %>%
  add_min_set_objective() %>%
  add_relative_targets(0.1) %>%
  add_binary_decisions() %>%
  add_cbc_solver(gap = 0.1, verbose = FALSE)

# generate solution %>%
s <- solve(p)

# plot solution
plot(s, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

add_connectivity_penalties *Add connectivity penalties*

Description

Add penalties to a conservation planning problem() to favor solutions that select planning units with high connectivity between them.

Usage

```
## S4 method for signature 'ConservationProblem,ANY,ANY,matrix'
add_connectivity_penalties(x, penalty, zones, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,Matrix'
add_connectivity_penalties(x, penalty, zones, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,data.frame'
add_connectivity_penalties(x, penalty, zones, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,dgCMatrix'
add_connectivity_penalties(x, penalty, zones, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,array'
add_connectivity_penalties(x, penalty, zones, data)
```

Arguments

x problem() (i.e. ConservationProblem) object.

penalty numeric penalty that is used to scale the importance of selecting planning units with strong connectivity between them compared to the main problem objective (e.g. solution cost when the argument to x has a minimum set objective set using add_min_set_objective()). Higher penalty values can be used to obtain solutions with a high degree of connectivity, and smaller penalty values can be used to obtain solutions with a small degree of connectivity. Note that negative penalty values can be used to obtain solutions that have very little connectivity.

zones matrix or Matrix object describing the level of connectivity between different zones. Each row and column corresponds to a different zone in the argument to x, and cell values indicate the level of connectivity between each combination of zones. Cell values along the diagonal of the matrix represent the level of connectivity between planning units allocated to the same zone. Cell values must lay between 1 and -1, where negative values favor solutions with weak connectivity. The default argument to zones is an identity matrix (i.e. a matrix with ones along the matrix diagonal and zeros elsewhere), so that planning units are only considered to be connected when they are allocated to the same zone. This argument is required when the argument to data is a matrix or Matrix object. If the argument to data is an array or data.frame with zone data, this argument must explicitly be set to NULL otherwise an error will be thrown.

data matrix, Matrix, data.frame, or array object containing connectivity data. The connectivity values correspond to the strength of connectivity between different planning units. Thus connections between planning units that are associated with higher values are more favorable in the solution. See the Data format section for more information.

Details

This function adds penalties to conservation planning problem to penalize solutions that have low connectivity. Specifically, it **favors** pair-wise connections between planning units that have high connectivity values. It was inspired by Beger *et al.* (2010) and can symmetric and asymmetric connectivity relationships between planning units.

The connectivity penalties are calculated using the following equations. Let I represent the set of planning units (indexed by i or j), Z represent the set of management zones (indexed by z or y), and X_{iz} represent the decision variable for planning unit i for in zone z (e.g. with binary values one indicating if planning unit is allocated or not). Also, let p represent the argument to penalty, D represent the argument to data, and W represent the argument to zones.

If the argument to data is supplied as a matrix or Matrix object, then the penalties are calculated as:

$$\sum_{i}^{I}\sum_{j}^{I}\sum_{z}^{Z}\sum_{y}^{Z}(-p \times X_{iz} \times X_{jy} \times D_{ij} \times W_{zy})$$

Otherwise, if the argument to data is supplied as a data.frame or array object, then the penalties are calculated as:

$$\sum_{i}^{I}\sum_{j}^{I}\sum_{z}^{Z}\sum_{y}^{Z}(-p \times X_{iz} \times X_{jy} \times D_{ijzy})$$

Note that when the problem objective is to maximize some measure of benefit and not minimize some measure of cost, the term $-p$ is replaced with p.

Value

Object (i.e. ConservationProblem) with the penalties added to it.

Data format

The argument to data can be specified using several different formats. These formats can be used to describe symmetric or asymmetric relationships between planning units.

matrix, Matrix where rows and columns represent different planning units and the value of each cell represents the strength of connectivity between two different planning units. Cells that occur along the matrix diagonal are treated as weights which indicate that planning units are more desirable in the solution. The argument to zones can be used to control the strength of connectivity between planning units in different zones. The default argument for zones is to treat planning units allocated to different zones as having zero connectivity.

`data.frame` containing the fields (columns) `"id1"`, `"id2"`, and `"boundary"`. Here, each row denotes the connectivity between two planning units following the *Marxan* format. The data can be used to denote symmetric or asymmetric relationships between planning units. By default, input data is assumed to be symmetric unless asymmetric data is also included (e.g. if data is present for planning units 2 and 3, then the same amount of connectivity is expected for planning units 3 and 2, unless connectivity data is also provided for planning units 3 and 2). If the argument to `x` contains multiple zones, then the columns `"zone1"` and `"zone2"` can optionally be provided to manually specify the connectivity values between planning units when they are allocated to specific zones. If the columns `"zone1"` and `"zone2"` are present, then the argument to `zones` must be `NULL`.

`array` containing four-dimensions where cell values indicate the strength of connectivity between planning units when they are assigned to specific management zones. The first two dimensions (i.e. rows and columns) indicate the strength of connectivity between different planning units and the second two dimensions indicate the different management zones. Thus the `data[1,2,3,4]` indicates the strength of connectivity between planning unit 1 and planning unit 2 when planning unit 1 is assigned to zone 3 and planning unit 2 is assigned to zone 4.

References

Beger M, Linke S, Watts M, Game E, Treml E, Ball I, and Possingham, HP (2010) Incorporating asymmetric connectivity into spatial decision making for conservation, *Conservation Letters*, 3: 359–368.

See Also

penalties.

Examples

```
# set seed for reproducibility
set.seed(600)

# load Matrix package for visualizing matrices
require(Matrix)

# load data
data(sim_pu_polygons, sim_pu_zones_stack, sim_features, sim_features_zones)

# define function to rescale values between zero and one so that we
# can compare solutions from different connectivity matrices
rescale <- function(x, to = c(0, 1), from = range(x, na.rm = TRUE)) {
  (x - from[1]) / diff(from) * diff(to) + to[1]
}

# create basic problem
p1 <- problem(sim_pu_polygons, sim_features, "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_default_solver(verbose = FALSE)

# create a symmetric connectivity matrix where the connectivity between
```

```
# two planning units corresponds to their shared boundary length
b_matrix <- boundary_matrix(sim_pu_polygons)

# standardize matrix values to lay between zero and one
b_matrix[] <- rescale(b_matrix[])

# visualize connectivity matrix
## Not run:
image(b_matrix)

## End(Not run)
# create a symmetric connectivity matrix where the connectivity between
# two planning units corresponds to their spatial proximity
# i.e. planning units that are further apart share less connectivity
centroids <- rgeos::gCentroid(sim_pu_polygons, byid = TRUE)
d_matrix <- (1 / (as(dist(centroids@coords), "Matrix") + 1))

# standardize matrix values to lay between zero and one
d_matrix[] <- rescale(d_matrix[])

# remove connections between planning units without connectivity to
# reduce run-time
d_matrix[d_matrix < 0.7] <- 0

# visualize connectivity matrix
## Not run:
image(d_matrix)

## End(Not run)
# create a symmetric connectivity matrix where the connectivity
# between adjacent two planning units corresponds to their combined
# value in a field in the planning unit attribute data
# for example, this field could describe the extent of native vegetation in
# each planning unit and we could use connectivity penalties to identify
# solutions that cluster planning units together that both contain large
# amounts of native vegetation
c_matrix <- connectivity_matrix(sim_pu_polygons, "cost")

# standardize matrix values to lay between zero and one
c_matrix[] <- rescale(c_matrix[])

# visualize connectivity matrix
## Not run:
image(c_matrix)

## End(Not run)
# create an asymmetric connectivity matrix. Here, connectivity occurs between
# adjacent planning units and, due to rivers flowing southwards
# through the study area, connectivity from northern planning units to
# southern planning units is ten times stronger than the reverse.
ac_matrix <- matrix(0, length(sim_pu_polygons), length(sim_pu_polygons))
ac_matrix <- as(ac_matrix, "Matrix")
adjacent_units <- rgeos::gIntersects(sim_pu_polygons, byid = TRUE)
```

```
for (i in seq_len(length(sim_pu_polygons))) {
  for (j in seq_len(length(sim_pu_polygons))) {
    # find if planning units are adjacent
    if (adjacent_units[i, j]) {
      # find if planning units lay north and south of each other
      # i.e. they have the same x-coordinate
      if (centroids@coords[i, 1] == centroids@coords[j, 1]) {
        if (centroids@coords[i, 2] > centroids@coords[j, 2]) {
          # if i is north of j add 10 units of connectivity
          ac_matrix[i, j] <- ac_matrix[i, j] + 10
        } else if (centroids@coords[i, 2] < centroids@coords[j, 2]) {
          # if i is south of j add 1 unit of connectivity
          ac_matrix[i, j] <- ac_matrix[i, j] + 1
        }
      }
    }
  }
}

# standardize matrix values to lay between zero and one
ac_matrix[] <- rescale(ac_matrix[])

# visualize asymmetric connectivity matrix
## Not run:
image(ac_matrix)

## End(Not run)
# create penalties
penalties <- c(10, 25)

# create problems using the different connectivity matrices and penalties
p2 <- list(p1,
           p1 %>% add_connectivity_penalties(penalties[1], data = b_matrix),
           p1 %>% add_connectivity_penalties(penalties[2], data = b_matrix),
           p1 %>% add_connectivity_penalties(penalties[1], data = d_matrix),
           p1 %>% add_connectivity_penalties(penalties[2], data = d_matrix),
           p1 %>% add_connectivity_penalties(penalties[1], data = c_matrix),
           p1 %>% add_connectivity_penalties(penalties[2], data = c_matrix),
           p1 %>% add_connectivity_penalties(penalties[1], data = ac_matrix),
           p1 %>% add_connectivity_penalties(penalties[2], data = ac_matrix))

# assign names to the problems
names(p2) <- c("basic problem",
               paste0("b_matrix (", penalties,")"),
               paste0("d_matrix (", penalties,")"),
               paste0("c_matrix (", penalties,")"),
               paste0("ac_matrix (", penalties,")"))
## Not run:
# solve problems
s2 <- lapply(p2, solve)

# plot solutions
par(mfrow = c(3, 3))
```

```
for (i in seq_along(s2)) {
  plot(s2[[i]], main = names(p2)[i], cex = 1.5, col = "white")
  plot(s2[[i]][s2[[i]]$solution_1 == 1, ], col = "darkgreen", add = TRUE)
}

## End(Not run)

# create minimal multi-zone problem and limit solver to one minute
# to obtain solutions in a short period of time
p3 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(0.15, nrow = 5, ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(time_limit = 60, verbose = FALSE)

# create matrix showing which planning units are adjacent to other units
a_matrix <- adjacency_matrix(sim_pu_zones_stack)

# visualize matrix
## Not run:
image(a_matrix)

## End(Not run)
# create a zone matrix where connectivities are only present between
# planning units that are allocated to the same zone
zm1 <- as(diag(3), "Matrix")

# print zone matrix
print(zm1)

# create a zone matrix where connectivities are strongest between
# planning units allocated to different zones
zm2 <- matrix(1, ncol = 3, nrow = 3)
diag(zm2) <- 0
zm2 <- as(zm2, "Matrix")

# print zone matrix
print(zm2)

# create a zone matrix that indicates that connectivities between planning
# units assigned to the same zone are much higher than connectivities
# assigned to different zones
zm3 <- matrix(0.1, ncol = 3, nrow = 3)
diag(zm3) <- 1
zm3 <- as(zm3, "Matrix")

# print zone matrix
print(zm3)

# create a zone matrix that indicates that connectivities between planning
# units allocated to zone 1 are very high, connectivities between planning
# units allocated to zones 1 and 2 are moderately high, and connectivities
# planning units allocated to other zones are low
```

```
zm4 <- matrix(0.1, ncol = 3, nrow = 3)
zm4[1, 1] <- 1
zm4[1, 2] <- 0.5
zm4[2, 1] <- 0.5
zm4 <- as(zm4, "Matrix")

# print zone matrix
print(zm4)

# create a zone matrix with strong connectivities between planning units
# allocated to the same zone, moderate connectivities between planning
# unit allocated to zone 1 and zone 2, and negative connectivities between
# planning units allocated to zone 3 and the other two zones
zm5 <- matrix(-1, ncol = 3, nrow = 3)
zm5[1, 2] <- 0.5
zm5[2, 1] <- 0.5
diag(zm5) <- 1
zm5 <- as(zm5, "Matrix")

# print zone matrix
print(zm5)

# create vector of penalties to use creating problems
penalties2 <- c(5, 15)

# create multi-zone problems using the adjacent connectivity matrix and
# different zone matrices
p4 <- list(
  p3,
  p3 %>% add_connectivity_penalties(penalties2[1], zm1, a_matrix),
  p3 %>% add_connectivity_penalties(penalties2[2], zm1, a_matrix),
  p3 %>% add_connectivity_penalties(penalties2[1], zm2, a_matrix),
  p3 %>% add_connectivity_penalties(penalties2[2], zm2, a_matrix),
  p3 %>% add_connectivity_penalties(penalties2[1], zm3, a_matrix),
  p3 %>% add_connectivity_penalties(penalties2[2], zm3, a_matrix),
  p3 %>% add_connectivity_penalties(penalties2[1], zm4, a_matrix),
  p3 %>% add_connectivity_penalties(penalties2[2], zm4, a_matrix),
  p3 %>% add_connectivity_penalties(penalties2[1], zm5, a_matrix),
  p3 %>% add_connectivity_penalties(penalties2[2], zm5, a_matrix))

# assign names to the problems
names(p4) <- c("basic problem",
               paste0("zm", rep(seq_len(5), each = 2), " (",
                      rep(penalties2, 2), ")"))
## Not run:
# solve problems
s4 <- lapply(p4, solve)
s4 <- lapply(s4, category_layer)
s4 <- stack(s4)

# plot solutions
plot(s4, main = names(p4), axes = FALSE, box = FALSE)
```

```
## End(Not run)

# create an array to manually specify the connectivities between
# each planning unit when they are allocated to each different zone
# for real-world problems, these connectivities would be generated using
# data - but here these connectivity values are assigned as random
# ones or zeros
c_array <- array(0, c(rep(ncell(sim_pu_zones_stack[[1]]), 2), 3, 3))
for (z1 in seq_len(3))
  for (z2 in seq_len(3))
    c_array[, , z1, z2] <- round(runif(ncell(sim_pu_zones_stack[[1]]) ^ 2,
                                       0, 0.505))

# create a problem with the manually specified connectivity array
# note that the zones argument is set to NULL because the connectivity
# data is an array
p5 <- list(p3,
           p3 %>% add_connectivity_penalties(15, zones = NULL, c_array))


# assign names to the problems
names(p5) <- c("basic problem", "connectivity array")
## Not run:
# solve problems
s5 <- lapply(p5, solve)
s5 <- lapply(s5, category_layer)
s5 <- stack(s5)

# plot solutions
plot(s5, main = names(p5), axes = FALSE, box = FALSE)

## End(Not run)
```

---

add_contiguity_constraints *Add contiguity constraints*

Description

Add constraints to a conservation planning problem() to ensure that all selected planning units are spatially connected with each other and form a single contiguous unit.

Usage

```
## S4 method for signature 'ConservationProblem,ANY,ANY'
add_contiguity_constraints(x, zones, data)

## S4 method for signature 'ConservationProblem,ANY,data.frame'
add_contiguity_constraints(x, zones, data)
```

```
## S4 method for signature 'ConservationProblem,ANY,matrix'
add_contiguity_constraints(x, zones, data)
```

Arguments

x problem() (i.e. ConservationProblem) object.

zones matrix or Matrix object describing the connection scheme for different zones. Each row and column corresponds to a different zone in the argument to x, and cell values must contain binary numeric values (i.e. one or zero) that indicate if connected planning units (as specified in the argument to data) should be still considered connected if they are allocated to different zones. The cell values along the diagonal of the matrix indicate if planning units should be subject to contiguity constraints when they are allocated to a given zone. Note arguments to zones must be symmetric, and that a row or column has a value of one then the diagonal element for that row or column must also have a value of one. The default argument to zones is an identity matrix (i.e. a matrix with ones along the matrix diagonal and zeros elsewhere), so that planning units are only considered connected if they are both allocated to the same zone.

data NULL, matrix, Matrix, data.frame object showing which planning units are connected with each other. The argument defaults to NULL which means that the connection data is calculated automatically using the adjacency_matrix() function. See the Data format section for more information.

Details

This function uses connection data to identify solutions that form a single contiguous unit. It was inspired by the mathematical formulations detailed in Önal and Briers (2006).

Value

Object (i.e. ConservationProblem) with the constraints added to it.

Data format

The argument to data can be specified using the following formats.

NULL connection data should be calculated automatically using the adjacency_matrix() function. This is the default argument. Note that the connection data must be manually defined using one of the other formats below when the planning unit data in the argument to x is not spatially referenced (e.g. in data.frame or numeric format).

matrix**,** Matrix where rows and columns represent different planning units and the value of each cell indicates if the two planning units are connected or not. Cell values should be binary numeric values (i.e. one or zero). Cells that occur along the matrix diagonal have no effect on the solution at all because each planning unit cannot be a connected with itself.

data.frame containing the fields (columns) "id1", "id2", and "boundary". Here, each row denotes the connectivity between two planning units following the *Marxan* format. The field boundary should contain binary numeric values that indicate if the two planning units specified in the fields "id1" and "id2" are connected or not. This data can be used to describe symmetric or asymmetric relationships between planning units. By default, input data is assumed to be symmetric unless asymmetric data is also included (e.g. if data is present for planning units 2 and 3, then the same amount of connectivity is expected for planning units 3 and 2, unless connectivity data is also provided for planning units 3 and 2).

Notes

In early versions, this function was named as the `add_connected_constraints()` function.

References

Önal H and Briers RA (2006) Optimal selection of a connected reserve network. *Operations Research*, 54: 379–388.

See Also

constraints.

Examples

```
# load data
data(sim_pu_raster, sim_features, sim_pu_zones_stack, sim_features_zones)

# create minimal problem
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create problem with added connected constraints
p2 <- p1 %>% add_contiguity_constraints()
## Not run:
# solve problems
s <- stack(solve(p1), solve(p2))

# plot solutions
plot(s, main = c("basic solution", "connected solution"), axes = FALSE,
     box = FALSE)

## End(Not run)
# create minimal problem with multiple zones, and limit the solver to
# 30 seconds to obtain solutions in a feasible period of time
p3 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(0.2, ncol = 3, nrow = 5)) %>%
      add_binary_decisions() %>%
      add_default_solver(time_limit = 30, verbose = FALSE)

# create problem with added constraints to ensure that the planning units
# allocated to each zone form a separate contiguous unit
z4 <- diag(3)
```

```
print(z4)
p4 <- p3 %>% add_contiguity_constraints(z4)

# create problem with added constraints to ensure that the planning
# units allocated to each zone form a separate contiguous unit,
# except for planning units allocated to zone 2 which do not need
# form a single contiguous unit
z5 <- diag(3)
z5[3, 3] <- 0
print(z5)
p5 <- p3 %>% add_contiguity_constraints(z5)

# create problem with added constraints that ensure that the planning
# units allocated to zones 1 and 2 form a contiguous unit
z6 <- diag(3)
z6[1, 2] <- 1
z6[2, 1] <- 1
print(z6)
p6 <- p3 %>% add_contiguity_constraints(z6)
## Not run:
# solve problems
s2 <- lapply(list(p3, p4, p5, p6), solve)
s2 <- lapply(s2, category_layer)
s2 <- stack(s2)

# plot solutions
plot(s2, axes = FALSE, box = FALSE,
     main = c("basic solution", "p4", "p5", "p6"))

## End(Not run)
# create a problem that has a main "reserve zone" and a secondary
# "corridor zone" to connect up import areas. Here, each feature has a
# target of 30% of its distribution. If a planning unit is allocated to the
# "reserve zone", then the prioritization accrues 100% of the amount of
# each feature in the planning unit. If a planning unit is allocated to the
# "corridor zone" then the prioritization accrues 40% of the amount of each
# feature in the planning unit. Also, the cost of managing a planning unit
# in the "corridor zone" is 45% of that when it is managed as the
# "reserve zone". Finally, the problem has constraints which
# ensure that all of the selected planning units form a single contiguous
# unit, so that the planning units allocated to the "corridor zone" can
# link up the planning units allocated to the "reserve zone"

# create planning unit data
pus <- sim_pu_zones_stack[[c(1, 1)]]
pus[[2]] <- pus[[2]] * 0.45
print(pus)

# create biodiversity data
fts <- zones(sim_features, sim_features * 0.4,
             feature_names = names(sim_features),
             zone_names = c("reserve zone", "corridor zone"))
print(fts)
```

```
# create targets
targets <- tibble::tibble(feature = names(sim_features),
                          zone = list(zone_names(fts))[rep(1, 5)],
                          target = cellStats(sim_features, "sum") * 0.2,
                          type = rep("absolute", 5))
print(targets)

# create zones matrix
z7 <- matrix(1, ncol = 2, nrow = 2)
print(z7)

# create problem
p7 <- problem(pus, fts) %>%
      add_min_set_objective() %>%
      add_manual_targets(targets) %>%
      add_contiguity_constraints(z7) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problems
s7 <- category_layer(solve(p7))

# plot solutions
plot(s7, "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

---

add_cplex_solver *Add a* CPLEX *solver*

Description

Specify that the *IBM CPLEX* software (IBM 2017) should be used to solve a conservation planning `problem()`. This function can also be used to customize the behavior of the solver. It requires the **cplexAPI** package to be installed (see below for installation instructions).

Usage

```
add_cplex_solver(
  x,
  gap = 0.1,
  time_limit = .Machine$integer.max,
  presolve = TRUE,
  threads = 1,
  verbose = TRUE
)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) 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 0.1 (i.e. 10% from optimality).
time_limit	`numeric` time limit (seconds) for generating solutions. The solver will return the current best solution when this time limit is exceeded. The default value is the largest integer value (i.e. `.Machine$integer.max`), effectively meaning that solver will keep running until a solution within the optimality gap is found.
presolve	`logical` attempt to simplify the problem before solving it? Defaults to `TRUE`.
threads	`integer` number of threads to use for the optimization algorithm. The default value is 1.
verbose	`logical` should information be printed while solving optimization problems? Defaults to `TRUE`.

Details

IBM CPLEX is a commercial optimization software. It is faster than the available open source solvers (e.g. `add_lpsymphony_solver()` and `add_rsymphony_solver()`. Although formal benchmarks examining the performance of this solver for conservation planning problems have yet to be completed, preliminary analyses suggest that it performs slightly slower than the *Gurobi* solver (i.e. `add_gurobi_solver()`). We recommend using this solver if the *Gurobi* solver is not available. Licenses are available for the *IBM CPLEX* software to academics at no cost (see `https://www.ibm.com/products/ilog-cplex-optimization-studio`).

Value

Object (i.e. `ConservationProblem`) with the solver added to it.

Installation

The pkgcplexAPI package is used to interface with *IBM CPLEX*. To install this package, the `CPLEX_BIN` variable must be set (similar to the `GUROBI_HOME` variable for the *Gurobi* software) to specify the file path for the *CPLEX* software. For example, on a Linux system, this variable can be specified by adding the following text to the ~/.bashrc file:

```
export CPLEX_BIN="/opt/ibm/ILOG/CPLEX_Studio128/cplex/bin/x86-64_linux/cplex"
```

Note that you may need to change the version number in the file path (i.e. `"CPLEX_Studio128"`). For more information on installing the pkgcplexAPI package, please see the official installation instructions for the package.

References

IBM (2017) IBM ILOG CPLEX Optimization Studio CPLEX User's Manual. Version 12 Release 8. IBM ILOG CPLEX Division, Incline Village, NV.

See Also

solvers.

Examples

```
## Not run:
# load data
data(sim_pu_raster, sim_features)

# create problem
p <- problem(sim_pu_raster, sim_features) %>%
     add_min_set_objective() %>%
     add_relative_targets(0.1) %>%
     add_binary_decisions() %>%
     add_cplex_solver(gap = 0.1, time_limit = 5, verbose = FALSE)

# generate solution
s <- solve(p)

# plot solution
plot(s, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

---

add_cuts_portfolio *Add Bender's cuts portfolio*

Description

Generate a portfolio of solutions for a conservation planning `problem()` using Bender's cuts (discussed in Rodrigues *et al.* 2000). This is recommended as a replacement for `add_gap_portfolio()` when the *Gurobi* software is not available.

Usage

```
add_cuts_portfolio(x, number_solutions = 10L)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object.
number_solutions	`integer` number of attempts to generate different solutions. Defaults to 10.

Details

This strategy for generating a portfolio of solutions involves solving the problem multiple times and adding additional constraints to forbid previously obtained solutions. In general, this strategy is most useful when problems take a long time to solve and benefit from having multiple threads allocated for solving an individual problem.

Value

Object (i.e. ConservationProblem) with the portfolio added to it.

Notes

In early versions (< 4.0.1), this function was only compatible with *Gurobi* (i.e. add_gurobi_solver()). To provide functionality with exact algorithm solvers, this function now adds constraints to the problem formulation to generate multiple solutions.

References

Rodrigues AS, Cerdeira OJ, and Gaston KJ (2000) Flexibility, efficiency, and accountability: adapting reserve selection algorithms to more complex conservation problems. *Ecography*, 23: 565–574.

See Also

portfolios.

Examples

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

# load data
data(sim_pu_raster, sim_features, sim_pu_zones_stack, sim_features_zones)

# create minimal problem with cuts portfolio
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_cuts_portfolio(10) %>%
      add_default_solver(gap = 0.2, verbose = FALSE)

## Not run:
# solve problem and generate 10 solutions within 20% of optimality
s1 <- solve(p1)

# plot solutions in portfolio
plot(stack(s1), axes = FALSE, box = FALSE)

## End(Not run)
# build multi-zone conservation problem with cuts portfolio
p2 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_binary_decisions() %>%
      add_cuts_portfolio(10) %>%
      add_default_solver(gap = 0.2, verbose = FALSE)

## Not run:
```

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

# print solution
str(s2, max.level = 1)

# plot solutions in portfolio
plot(stack(lapply(s2, category_layer)), main = "solution", axes = FALSE,
     box = FALSE)

## End(Not run)
```

add_default_decisions *Add default decisions*

Description

This function adds the default decision types to a conservation planning `problem()`. The default types are binary and are added using the `add_binary_decisions()` function.

Usage

```
add_default_decisions(x)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object.

See Also

decisions.

add_default_solver *Add a default solver*

Description

Identify the best solver currently installed on the system and specify that it should be used to solve a conservation planning `problem()`. For information on the performance of different solvers, please see Schuster *et al.* (2020) for benchmarks comparing the run time and solution quality of some of the available solvers when applied to different sized datasets.

Usage

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

Arguments

x	problem() (i.e. ConservationProblem) object.
...	arguments passed to the solver.

Details

Ranked from best to worst, the available solvers that can be used are: add_gurobi_solver(), add_cplex_solver(), add_cbc_solver(), add_lpsymphony_solver(), and finally add_rsymphony_solver().

Value

Object (i.e. ConservationProblem) with the solver added to it.

References

Schuster R, Hanson JO, Strimas-Mackey M, and Bennett JR (2020). Exact integer linear programming solvers outperform simulated annealing for solving conservation planning problems. *PeerJ*, 8: e9258.

See Also

solvers.

---

add_extra_portfolio *Add an extra portfolio*

Description

Generate a portfolio of solutions for a conservation planning problem() by storing feasible solutions discovered during the optimization process. This method is useful for quickly obtaining multiple solutions, but does not provide any guarantees on the number of solutions, or the quality of solutions.

Usage

```
add_extra_portfolio(x)
```

Arguments

x	problem() (i.e. ConservationProblem) object.

Details

This strategy for generating a portfolio requires problems to be solved using the *Gurobi* software suite (i.e. using add_gurobi_solver(). Specifically, version 8.0.0 (or greater) of the **gurobi** package must be installed.

Value

Object (i.e. ConservationProblem) with the portfolio added to it.

See Also

portfolios.

Examples

```
## Not run:
# set seed for reproducibility
set.seed(600)

# load data
data(sim_pu_raster, sim_features)

# create minimal problem with a portfolio for extra solutions
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.05) %>%
      add_extra_portfolio() %>%
      add_default_solver(gap = 0, verbose = FALSE)

# solve problem and generate portfolio
s1 <- solve(p1)

# print number of solutions found
print(length(s1))

# plot solutions
plot(stack(s1), axes = FALSE, box = FALSE)

# create multi-zone problem with a portfolio for extra solutions
p2 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_extra_portfolio() %>%
      add_default_solver(gap = 0, verbose = FALSE)

# solve problem and generate portfolio
s2 <- solve(p2)

# print number of solutions found
print(length(s2))

# plot solutions in portfolio
plot(stack(lapply(s2, category_layer)), main = "solution", axes = FALSE,
     box = FALSE)

## End(Not run)
```

add_feature_contiguity_constraints
Add feature contiguity constraints

Description

Add constraints to a problem() to ensure that each feature is represented in a contiguous unit of dispersible habitat. These constraints are a more advanced version of those implemented in the add_contiguity_constraints() function, because they ensure that each feature is represented in a contiguous unit and not that the entire solution should form a contiguous unit. Additionally, this function can use data showing the distribution of dispersible habitat for each feature to ensure that all features can disperse through out the areas designated for their conservation.

Usage

```
## S4 method for signature 'ConservationProblem,ANY,Matrix'
add_feature_contiguity_constraints(x, zones, data)

## S4 method for signature 'ConservationProblem,ANY,data.frame'
add_feature_contiguity_constraints(x, zones, data)

## S4 method for signature 'ConservationProblem,ANY,matrix'
add_feature_contiguity_constraints(x, zones, data)

## S4 method for signature 'ConservationProblem,ANY,ANY'
add_feature_contiguity_constraints(x, zones, data)
```

Arguments

x — problem() (i.e. ConservationProblem) object.

zones — matrix, Matrix or list object describing the connection scheme for different zones. For matrix or and Matrix arguments, each row and column corresponds to a different zone in the argument to x, and cell values must contain binary numeric values (i.e. one or zero) that indicate if connected planning units (as specified in the argument to data) should be still considered connected if they are allocated to different zones. The cell values along the diagonal of the matrix indicate if planning units should be subject to contiguity constraints when they are allocated to a given zone. Note arguments to zones must be symmetric, and that a row or column has a value of one then the diagonal element for that row or column must also have a value of one. If the connection scheme between different zones should differ among the features, then the argument to zones should be a list of matrix or Matrix objects that shows the specific scheme for each feature using the conventions described above. The default argument to zones is an identity matrix (i.e. a matrix with ones along the matrix diagonal and zeros elsewhere), so that planning units are only considered connected if they are both allocated to the same zone.

data — NULL, matrix, Matrix, data.frame or list of matrix, Matrix, or data.frame objects. The argument to data shows which planning units should be treated as being connected when implementing constraints to ensure that features are represented in contiguous units. If different features have different dispersal capabilities, then it may be desirable to specify which sets of planning units should be treated as being connected for which features using a list of objects. The default argument is NULL which means that the connection data is calculated automatically using the adjacency_matrix() function and so all adjacent planning units are treated as being connected for all features. See the Data format section for more information.

Details

This function uses connection data to identify solutions that represent features in contiguous units of dispersible habitat. It was inspired by the mathematical formulations detailed in Önal and Briers (2006) and Cardeira *et al.* 2010. For an example that has used these constraints, see Hanson *et al.* (2019). Please note that these constraints require the expanded formulation and therefore cannot be used with feature data that have negative vales. **Please note that adding these constraints to a problem will drastically increase the amount of time required to solve it.**

Value

Object (i.e. ConservationProblem) with the constraints added to it.

Data format

The argument to data can be specified using the following formats.

NULL connection data should be calculated automatically using the adjacency_matrix() function. This is the default argument and means that all adjacent planning units are treated as potentially dispersible for all features. Note that the connection data must be manually defined using one of the other formats below when the planning unit data in the argument to x is not spatially referenced (e.g. in data.frame or numeric format).

matrix, Matrix where rows and columns represent different planning units and the value of each cell indicates if the two planning units are connected or not. Cell values should be binary numeric values (i.e. one or zero). Cells that occur along the matrix diagonal have no effect on the solution at all because each planning unit cannot be a connected with itself. Note that pairs of connected planning units are treated as being potentially dispersible for all features.

data.frame containing the fields (columns) "id1", "id2", and "boundary". Here, each row denotes the connectivity between two planning units following the *Marxan* format. The field boundary should contain binary numeric values that indicate if the two planning units specified in the fields "id1" and "id2" are connected or not. This data can be used to describe symmetric or asymmetric relationships between planning units. By default, input data is assumed to be symmetric unless asymmetric data is also included (e.g. if data is present for planning units 2 and 3, then the same amount of connectivity is expected for planning units 3 and 2, unless connectivity data is also provided for planning units 3 and 2). Note that pairs of connected planning units are treated as being potentially dispersible for all features.

`list` containing `matrix`, `Matrix`, or `data.frame` objects showing which planning units should be treated as connected for each feature. Each element in the `list` should correspond to a different feature (specifically, a different target in the problem), and should contain a `matrix`, `Matrix`, or `data.frame` object that follows the conventions detailed above.

Notes

In early versions, it was named as the `add_corridor_constraints` function.

References

Önal H and Briers RA (2006) Optimal selection of a connected reserve network. *Operations Research*, 54: 379–388.

Cardeira JO, Pinto LS, Cabeza M and Gaston KJ (2010) Species specific connectivity in reserve-network design using graphs. *Biological Conservation*, 2: 408–415.

Hanson JO, Fuller RA, & Rhodes JR (2019) Conventional methods for enhancing connectivity in conservation planning do not always maintain gene flow. *Journal of Applied Ecology*, 56: 913–922.

See Also

constraints.

Examples

```
# load data
data(sim_pu_raster, sim_pu_zones_stack, sim_features, sim_features_zones)

# create minimal problem
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.3) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create problem with contiguity constraints
p2 <- p1 %>% add_contiguity_constraints()

# create problem with constraints to represent features in contiguous
# units
p3 <- p1 %>% add_feature_contiguity_constraints()

# create problem with constraints to represent features in contiguous
# units that contain highly suitable habitat values
# (specifically in the top 1.5th percentile)
cm4 <- lapply(seq_len(nlayers(sim_features)), function(i) {
  # create connectivity matrix using the i'th feature's habitat data
  m <- connectivity_matrix(sim_pu_raster, sim_features[[i]])
  # convert matrix to TRUE/FALSE values in top 20th percentile
  m <- m > quantile(as.vector(m), 1 - 0.015, names = FALSE)
  # convert matrix from TRUE/FALSE to sparse matrix with 0/1s
  m <- as(m, "dgCMatrix")
```

```
  # remove 0s from the sparse matrix
  m <- Matrix::drop0(m)
  # return matrix
  m
})
p4 <- p1 %>% add_feature_contiguity_constraints(data = cm4)
## Not run:
# solve problems
s1 <- stack(solve(p1), solve(p2), solve(p3), solve(p4))

# plot solutions
plot(s1,  axes = FALSE, box = FALSE,
     main = c("basic solution", "contiguity constraints",
              "feature contiguity constraints",
              "feature contiguity constraints with data"))

## End(Not run)
# create minimal problem with multiple zones, and limit the solver to
# 30 seconds to obtain solutions in a feasible period of time
p5 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(0.1, ncol = 3, nrow = 5)) %>%
      add_binary_decisions() %>%
      add_default_solver(time_limit = 30, verbose = FALSE)

# create problem with contiguity constraints that specify that the
# planning units used to conserve each feature in different management
# zones must form separate contiguous units
p6 <- p5 %>% add_feature_contiguity_constraints(diag(3))

# create problem with contiguity constraints that specify that the
# planning units used to conserve each feature must form a single
# contiguous unit if the planning units are allocated to zones 1 and 2
# and do not need to form a single contiguous unit if they are allocated
# to zone 3
zm7 <- matrix(0, ncol = 3, nrow = 3)
zm7[seq_len(2), seq_len(2)] <- 1
print(zm7)
p7 <- p5 %>% add_feature_contiguity_constraints(zm7)

# create problem with contiguity constraints that specify that all of
# the planning units in all three of the zones must conserve first feature
# in a single contiguous unit but the planning units used to conserve the
# remaining features do not need to be contiguous in any way
zm8 <- lapply(seq_len(number_of_features(sim_features_zones)), function(i)
  matrix(ifelse(i == 1, 1, 0), ncol = 3, nrow = 3))
print(zm8)
p8 <- p5 %>% add_feature_contiguity_constraints(zm8)
## Not run:
# solve problems
s2 <- lapply(list(p5, p6, p7, p8), solve)
s2 <- stack(lapply(s2, category_layer))
```

```
# plot solutions
plot(s2, main = c("p5", "p6", "p7", "p8"), axes = FALSE, box = FALSE)

## End(Not run)
```

`add_feature_weights` *Add feature weights*

Description

Conservation planning problems that aim to maximize the representation of features given a budget often will not able to conserve all of the features unless the budget is very high. In such budget-limited problems, it may be desirable to prefer the representation of some features over other features. This information can be incorporated into the problem using weights. Weights can be applied to a problem to favor the representation of some features over other features when making decisions about how the budget should be allocated.

Usage

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

## S4 method for signature 'ConservationProblem,matrix'
add_feature_weights(x, weights)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object.
weights	`numeric` or `matrix` of weights. See the Weights format section for more information.

Details

Weights can only be applied to problems that have an objective that is budget limited (e.g. `add_max_cover_objective()`). They can be applied to problems that aim to maximize phylogenetic representation (`add_max_phylo_div_objective()`) to favor the representation of specific features over the representation of some phylogenetic branches. Weights cannot be negative values and must have values that are equal to or larger than zero. **Note that planning unit costs are scaled to 0.01 to identify the cheapest solution among multiple optimal solutions. This means that the optimization process will favor cheaper solutions over solutions that meet feature targets (or occurrences) when feature weights are lower than 0.01.**

Value

Object (i.e. `ConservationProblem`) with the weights added to it.

Weights format

The argument to `weights` can be specified using the following formats.

`numeric` containing weights for each feature. Note that this type of argument cannot be used to specify weights for problems with multiple zones.

`matrix` containing weights for each feature in each zone. Here, each row corresponds to a different feature in argument to `x`, each column corresponds to a different zone in argument to `x`, and each cell contains the weight value for a given feature that the solution can to secure in a given zone. Note that if the problem contains targets created using `add_manual_targets()` then a `matrix` should be supplied containing a single column that indicates that weight for fulfilling each target.

See Also

objectives.

Examples

```
# load ape package
require(ape)

# load data
data(sim_pu_raster, sim_features, sim_phylogeny, sim_pu_zones_stack,
     sim_features_zones)

# create minimal problem that aims to maximize the number of features
# adequately conserved given a total budget of 3800. Here, each feature
# needs 20% of its habitat for it to be considered adequately conserved
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_max_features_objective(budget = 3800) %>%
      add_relative_targets(0.2) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create weights that assign higher importance to features with less
# suitable habitat in the study area
(w2 <- exp((1 / cellStats(sim_features, "sum")) * 200))

# create problem using rarity weights
p2 <- p1 %>% add_feature_weights(w2)

# create manually specified weights that assign higher importance to
# certain features. These weights could be based on a pre-calculated index
# (e.g. an index measuring extinction risk where higher values
# denote higher extinction risk)
w3 <- c(0, 0, 0, 100, 200)
p3 <- p1 %>% add_feature_weights(w3)
## Not run:
# solve problems
s1 <- stack(solve(p1), solve(p2), solve(p3))
```

```
# plot solutions
plot(s1, main = c("equal weights", "rarity weights", "manual weights"),
     axes = FALSE, box = FALSE)

## End(Not run)

# plot the example phylogeny
## Not run:
par(mfrow = c(1, 1))
plot(sim_phylogeny, main = "simulated phylogeny")

## End(Not run)
# create problem with a maximum phylogenetic diversity objective,
# where each feature needs 10% of its distribution to be secured for
# it to be adequately conserved and a total budget of 1900
p4 <- problem(sim_pu_raster, sim_features) %>%
      add_max_phylo_div_objective(1900, sim_phylogeny) %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s4 <- solve(p4)

# plot solution
plot(s4, main = "solution", axes = FALSE, box = FALSE)

# find out which features have their targets met
r4 <- eval_target_coverage_summary(p4, s4)
print(r4, width = Inf)

# plot the example phylogeny and color the represented features in red
plot(sim_phylogeny, main = "represented features",
     tip.color = replace(rep("black", nlayers(sim_features)),
                         which(r4$met), "red"))

## End(Not run)
# we can see here that the third feature ("layer.3", i.e.
# sim_features[[3]]) is not represented in the solution. Let us pretend
# that it is absolutely critical this feature is adequately conserved
# in the solution. For example, this feature could represent a species
# that plays important role in the ecosystem, or a species that is
# important commercial activities (e.g. eco-tourism). So, to generate
# a solution that conserves the third feature whilst also aiming to
# maximize phylogenetic diversity, we will create a set of weights that
# assign a particularly high weighting to the third feature
w5 <- c(0, 0, 1000, 0, 0)

# we can see that this weighting (i.e. w5[3]) has a much higher value than
# the branch lengths in the phylogeny so solutions that represent this
# feature be much closer to optimality
print(sim_phylogeny$edge.length)
## Not run:
```

```
# create problem with high weighting for the third feature and solve it
s5 <- p4 %>% add_feature_weights(w5) %>% solve()

# plot solution
plot(s5, main = "solution", axes = FALSE, box = FALSE)

# find which features have their targets met
r5 <- eval_target_coverage_summary(p4, s5)
print(r5, width = Inf)

# plot the example phylogeny and color the represented features in red
# here we can see that this solution only adequately conserves the
# third feature. This means that, given the budget, we are faced with the
# trade-off of conserving either the third feature, or a phylogenetically
# diverse set of three different features.
plot(sim_phylogeny, main = "represented features",
     tip.color = replace(rep("black", nlayers(sim_features)),
                         which(r5$met), "red"))

## End(Not run)
# create multi-zone problem with maximum features objective,
# with 10% representation targets for each feature, and set
# a budget such that the total maximum expenditure in all zones
# cannot exceed 3000
p6 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_max_features_objective(3000) %>%
      add_relative_targets(matrix(0.1, ncol = 3, nrow = 5)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create weights that assign equal weighting for the representation
# of each feature in each zone except that it does not matter if
# feature 1 is represented in zone 1 and it really important
# that feature 3 is really in zone 1
w7 <- matrix(1, ncol = 3, nrow = 5)
w7[1, 1] <- 0
w7[3, 1] <- 100

# create problem with weights
p7 <- p6 %>% add_feature_weights(w7)
## Not run:
# solve problems
s6 <- solve(p6)
s7 <- solve(p7)

# plot solutions
plot(stack(category_layer(s6), category_layer(s7)),
     main = c("equal weights", "manual weights"), axes = FALSE, box = FALSE)

## End(Not run)
# create minimal problem to show the correct method for setting
# weights for problems with manual targets
p8 <- problem(sim_pu_raster, sim_features) %>%
```

```
      add_max_features_objective(budget = 3000) %>%
      add_manual_targets(data.frame(feature = c("layer.1", "layer.4"),
                                    type = "relative",
                                    target = 0.1)) %>%
      add_feature_weights(matrix(c(1, 200), ncol = 1)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s8 <- solve(p8)

# plot solution
plot(s8, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

add_gap_portfolio *Add a gap portfolio*

Description

Generate a portfolio of solutions for a conservation planning `problem()` by finding a certain number of solutions that are all within a pre-specified optimality gap. This method is useful for generating multiple solutions that can be used to calculate selection frequencies for moderate and large-sized problems (similar to *Marxan*).

Usage

```
add_gap_portfolio(x, number_solutions, pool_gap = 0.1)
```

Arguments

`x`	`problem()` (i.e. `ConservationProblem`) object.
`number_solutions`	`integer` number of solutions required.
`pool_gap`	`numeric` gap to optimality for solutions in the portfolio. This relative gap specifies a threshold worst-case performance for solutions in the portfolio. For example, value of 0.1 will result in the portfolio returning solutions that are within 10% of an optimal solution. Note that the gap specified in the solver (i.e. `add_gurobi_solver()` must be less than or equal to the gap specified to generate the portfolio. Defaults to 0.1.

Details

This strategy for generating a portfolio requires problems to be solved using the *Gurobi* software suite (i.e. using `add_gurobi_solver()`. Specifically, version 9.0.0 (or greater) of the **gurobi** package must be installed. Note that the number of solutions returned may be less than the argument to `number_solutions`, if the total number of solutions that meet the optimality gap is less than the number of solutions requested. Also, note that this portfolio function only works with problems that have binary decisions (i.e. specified using `add_binary_decisions()`).

Value

Object (i.e. ConservationProblem) with the portfolio added to it.

See Also

portfolios.

Examples

```
## Not run:
# set seed for reproducibility
set.seed(600)

# load data
data(sim_pu_raster, sim_features)

# create minimal problem with a portfolio containing 10 solutions within 20%
# of optimality
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.05) %>%
      add_gap_portfolio(number_solutions = 5, pool_gap = 0.2) %>%
      add_default_solver(gap = 0, verbose = FALSE)

# solve problem and generate portfolio
s1 <- solve(p1)

# print number of solutions found
print(length(s1))

# plot solutions
plot(stack(s1), axes = FALSE, box = FALSE)

# create multi-zone  problem with a portfolio containing 10 solutions within
# 20% of optimality
p2 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_gap_portfolio(number_solutions = 5, pool_gap = 0.2) %>%
      add_default_solver(gap = 0, verbose = FALSE)

# solve problem and generate portfolio
s2 <- solve(p2)

# print number of solutions found
print(length(s2))

# plot solutions in portfolio
plot(stack(lapply(s2, category_layer)), main = "solution", axes = FALSE,
     box = FALSE)
```

```
## End(Not run)
```

---

add_gurobi_solver *Add a* Gurobi *solver*

Description

Specify that the *Gurobi* software (Gurobi Optimization LLC 2021) should be used to solve a conservation planning `problem()`. This function can also be used to customize the behavior of the solver. It requires the **gurobi** package to be installed (see below for installation instructions).

Usage

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

Arguments

x	`problem()` (i.e. `ConservationProblem`) 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 0.1 (i.e. 10% from optimality).
time_limit	`numeric` time limit (seconds) for generating solutions. The solver will return the current best solution when this time limit is exceeded. The default value is the largest integer value (i.e. `.Machine$integer.max`), effectively meaning that solver will keep running until a solution within the optimality gap is found.
presolve	`integer` number indicating how intensively the solver should try to simplify the problem before solving it. Available options are: (-1) automatically determine the intensity of pre-solving, (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 is 1.

`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` `logical` should extra attention be paid to verifying the accuracy of numerical calculations? This may be useful when dealing problems that may suffer from numerical instability issues. Beware that it will likely substantially increase run time (sets the *Gurobi* `NumericFocus` parameter to 3). Defaults to `FALSE`.

`start_solution` `NULL` or object containing the starting solution for the solver. Defaults to `NULL` such that no starting solution is used. To specify a starting solution, the argument to `start_solution` should be in the same format as the planning units (i.e. a `NULL`, `numeric`, `matrix`, `data.frame`, `Raster`, `Spatial`, or `sf::sf()` object). See the Start solution format section for more information.

`verbose` `logical` should information be printed while solving optimization problems? 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 available for generating prioritizations, however, it 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. For information on the performance of different solvers, please see Schuster *et al.* (2020) for benchmarks comparing the run time and solution quality of different solvers when applied to different sized datasets.

Value

Object (i.e. `ConservationProblem`) with the solver added to it.

Installation

Please see the *Gurobi Installation Guide* vignette for details on installing the *Gurobi* software and the **gurobi** package. You can access this vignette online or using the following code:

```
vignette("gurobi_installation", package = "prioritizr")
```

Start solution format

Broadly speaking, the argument to `start_solution` must be in the same format as the planning unit data in the argument to `x`. Further details on the correct format are listed separately for each of the different planning unit data formats:

`x` **has** `numeric` **planning units** The argument to `start_solution` must be a `numeric` vector with each element corresponding to a different planning unit. It should have the same number of planning units as those in the argument to `x`. Additionally, any planning units missing cost (`NA`) values should also have missing (`NA`) values in the argument to `start_solution`.

x **has** matrix **planning units** The argument to start_solution must be a matrix vector with each row corresponding to a different planning unit, and each column correspond to a different management zone. It should have the same number of planning units and zones as those in the argument to x. Additionally, any planning units missing cost (NA) values for a particular zone should also have a missing (NA) values in the argument to start_solution.

x **has** Raster **planning units** The argument to start_solution be a Raster object where different grid cells (pixels) correspond to different planning units and layers correspond to a different management zones. It should have the same dimensionality (rows, columns, layers), resolution, extent, and coordinate reference system as the planning units in the argument to x. Additionally, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to start_solution.

x **has** data.frame **planning units** The argument to start_solution must be a data.frame with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if a data.frame object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with sf::sf() data). Additionally, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to start_solution.

x **has** Spatial **planning units** The argument to start_solution must be a Spatial object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the Spatial object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with sf::sf() data). Additionally, the argument to start_solution must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to start_solution.

x **has** sf::sf() **planning units** The argument to start_solution must be a sf::sf() object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the sf::sf() object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example). Additionally, the argument to start_solution must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to start_solution.

References

Gurobi Optimization LLC (2021) Gurobi Optimizer Reference Manual. https://www.gurobi.com.

Schuster R, Hanson JO, Strimas-Mackey M, and Bennett JR (2020). Exact integer linear programming solvers outperform simulated annealing for solving conservation planning problems. *PeerJ*, 8: e9258.

See Also

solvers.

Examples

```
## Not run:
# load data
data(sim_pu_raster, sim_features)

# create problem
p <- problem(sim_pu_raster, sim_features) %>%
     add_min_set_objective() %>%
     add_relative_targets(0.1) %>%
     add_binary_decisions() %>%
     add_gurobi_solver(gap = 0, verbose = FALSE)

# generate solution %>%
s <- solve(p)

# plot solution
plot(s, main = "solution", axes = FALSE, box = FALSE)

# create a similar problem with boundary length penalties and
# specify the solution from the previous run as a starting solution
p2 <- problem(sim_pu_raster, sim_features) %>%
     add_min_set_objective() %>%
     add_relative_targets(0.1) %>%
     add_boundary_penalties(10) %>%
     add_binary_decisions() %>%
     add_gurobi_solver(gap = 0, start_solution = s, verbose = FALSE)

# generate solution
s2 <- solve(p2)

# plot solution
plot(s2, main = "solution with boundary penalties", axes = FALSE,
     box = FALSE)

## End(Not run)
```

---

add_linear_penalties *Add linear penalties*

Description

Add penalties to a conservation planning `problem()` to penalize solutions that select planning units with higher values from a specific data source (e.g. anthropogenic impact). These penalties assume a linear trade-off between the penalty values and the primary objective of the conservation planning `problem()` (e.g. solution cost for minimum set problems; `add_min_set_objective()`.

Usage

```
## S4 method for signature 'ConservationProblem,ANY,character'
```

```
add_linear_penalties(x, penalty, data)

## S4 method for signature 'ConservationProblem,ANY,numeric'
add_linear_penalties(x, penalty, data)

## S4 method for signature 'ConservationProblem,ANY,matrix'
add_linear_penalties(x, penalty, data)

## S4 method for signature 'ConservationProblem,ANY,Matrix'
add_linear_penalties(x, penalty, data)

## S4 method for signature 'ConservationProblem,ANY,Raster'
add_linear_penalties(x, penalty, data)

## S4 method for signature 'ConservationProblem,ANY,dgCMatrix'
add_linear_penalties(x, penalty, data)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object.
penalty	`numeric` penalty value that is used to scale the importance not selecting planning units with high `data` values. Higher `penalty` values can be used to obtain solutions that are strongly averse to selecting places with high `data` values, and smaller `penalty` values can be used to obtain solutions that only avoid places with especially high `data` values. Note that negative `penalty` values can be used to obtain solutions that prefer places with high `data` values. Additionally, when adding these penalties to problems with multiple zones, the argument to `penalty` must have a value for each zone.
data	`character`, `numeric`, `Raster`, `matrix`, or `Matrix` object containing the data used to penalize solutions. Planning units that are associated with higher data values are penalized more strongly in the solution. See the Data format section for more information.

Details

This function penalizes solutions that have higher values according to the sum of the penalty values associated with each planning unit, weighted by status of each planning unit in the solution.

Specifically, the linear penalties are calculated using the following equations. Let I denote the set of planning units (indexed by i), Z the set of management zones (indexed by z), and X_{iz} the decision variable for allocating planning unit i to zone z (e.g. with binary values one indicating if planning unit is allocated or not). Also, let P_z represent the penalty scaling value for zones $z \in Z$ (argument to `penalty`), and D_{iz} the penalty data for allocating planning unit $i \in I$ to zones $z \in Z$ (argument to `data` if supplied as a `matrix` object).

$$\sum_{i}^{I} \sum_{z}^{Z} P_z \times D_{iz} \times X_{iz}$$

Note that when the problem objective is to maximize some measure of benefit and not minimize some measure of cost, the term P_z is replaced with $-P_z$.

Data format

The argument to data can be specified using the following formats.

character field (column) name(s) that contain the data for penalizing planning units. This type of argument is only compatible if the planning units in the argument to x are a Spatial, sf::sf(), or data.frame object. The fields (columns) must have numeric values, and must not contain any missing (NA) values. For problems involving multiple zones, the argument to data must contain a field name for each zone.

numeric vector containing the data for penalizing each planning unit. These values must not contain any missing (NA) values. Note that this type of argument is only available for planning units that contain a single zone.

Raster containing the data for penalizing planning units. This type of argument is only compatible if the planning units in the argument to x are Spatial, sf::sf(), or or Raster (i.e. they are in a spatially referenced format). If the planning unit data are a Spatial or sf::sf() object, then the penalty data are calculated by overlaying the planning units with the argument to data and calculating the sum of the values. If the planning unit data are in the Raster then the penalty data are calculated by extracting the cell values (note that the planning unit data and the argument to codedata must have exactly the same dimensionality, extent, and missingness). For problems involving multiple zones, the argument to data must contain a layer for each zone.

matrix, Matrix containing numeric values that specify data for penalizing each planning unit. Each row corresponds to a planning unit, each column corresponds to a zone, and each cell indicates the data for penalizing a planning unit when it is allocated to a given zone.

Examples

```
# set seed for reproducibility
set.seed(600)

# load data
data(sim_pu_polygons, sim_pu_zones_stack, sim_features, sim_features_zones)

# add a column to contain the penalty data for each planning unit
# e.g. these values could indicate the level of habitat
sim_pu_polygons$penalty_data <- runif(nrow(sim_pu_polygons))

# plot the penalty data to visualise its spatial distribution
spplot(sim_pu_polygons, zcol = "penalty_data", main = "penalty data",
       axes = FALSE, box = FALSE)

# create minimal problem with minimum set objective,
# this does not use the penalty data
p1 <- problem(sim_pu_polygons, sim_features, cost_column = "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# print problem
```

```
print(p1)

# create an updated version of the previous problem,
# with the penalties added to it
p2 <- p1 %>% add_linear_penalties(100, data = "penalty_data")

# print problem
print(p2)

## Not run:
# solve the two problems
s1 <- solve(p1)
s2 <- solve(p2)

# plot the solutions and compare them,
# since we supplied a very high penalty value (i.e. 100), relative
# to the range of values in the penalty data and the objective function,
# the solution in s2 is very sensitive to values in the penalty data
spplot(s1, zcol = "solution_1", main = "solution without penalties",
       axes = FALSE, box = FALSE)
spplot(s2, zcol = "solution_1", main = "solution with penalties",
       axes = FALSE, box = FALSE)

# for real conservation planning exercises,
# it would be worth exploring a range of penalty values (e.g. ranging
# from 1 to 100 increments of 5) to explore the trade-offs

## End(Not run)

# now, let's examine a conservation planning exercise involving multiple
# management zones

# create targets for each feature within each zone,
# these targets indicate that each zone needs to represent 10% of the
# spatial distribution of each feature
targ <- matrix(0.1, ncol = number_of_zones(sim_features_zones),
               nrow = number_of_features(sim_features_zones))

# create penalty data for allocating each planning unit to each zone,
# these data will be generated by simulating values
penalty_stack <- simulate_cost(sim_pu_zones_stack[[1]],
                               n = number_of_zones(sim_features_zones))

# plot the penalty data, each layer corresponds to a different zone
plot(penalty_stack, main = "penalty data", axes = FALSE, box = FALSE)

# create a multi-zone problem with the minimum set objective
# and penalties for allocating planning units to each zone,
# with a penalty scaling factor of 1 for each zone
p3 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_relative_targets(targ) %>%
      add_linear_penalties(c(1, 1, 1), penalty_stack) %>%
```

```
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# print problem
print(p3)

## Not run:
# solve problem
s3 <- solve(p3)

# plot solution
plot(category_layer(s3), main = "multi-zone solution",
     axes = FALSE, box = FALSE)

## End(Not run)
```

---

add_locked_in_constraints

Add locked in constraints

Description

Add constraints to a conservation planning problem() to ensure that specific planning units are selected (or allocated to a specific zone) in the solution. For example, it may be desirable to lock in planning units that are inside existing protected areas so that the solution fills in the gaps in the existing reserve network. If specific planning units should be locked out of a solution, use add_locked_out_constraints(). For problems with non-binary planning unit allocations (e.g. proportions), the add_manual_locked_constraints() function can be used to lock planning unit allocations to a specific value.

Usage

```
add_locked_in_constraints(x, locked_in)

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

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

## S4 method for signature 'ConservationProblem,matrix'
add_locked_in_constraints(x, locked_in)

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

## S4 method for signature 'ConservationProblem,Spatial'
add_locked_in_constraints(x, locked_in)
```

```
## S4 method for signature 'ConservationProblem,sf'
add_locked_in_constraints(x, locked_in)

## S4 method for signature 'ConservationProblem,Raster'
add_locked_in_constraints(x, locked_in)
```

Arguments

`x`	`problem()` (i.e. `ConservationProblem`) object.
`locked_in`	Object that determines which planning units that should be locked in. See the Data format section for more information.

Value

Object (i.e. `ConservationProblem`) with the constraints added to it.

Data format

The locked planning units can be specified using the following formats. Generally, the locked data should correspond to the planning units in the argument to `x`. To help make working with `Raster` planning unit data easier, the locked data should correspond to cell indices in the `Raster` data. For example, `integer` arguments should correspond to cell indices and `logical` arguments should have a value for each cell—regardless of which planning unit cells contain `NA` values.

`integer vector` of indices pertaining to which planning units should be locked for the solution. This argument is only compatible with problems that contain a single zone.

`logical vector` containing `TRUE` and/or `FALSE` values that indicate which planning units should be locked in the solution. This argument is only compatible with problems that contain a single zone.

`matrix` containing `logical` `TRUE` and/or `FALSE` values which indicate if certain planning units are should be locked to a specific zone in the solution. Each row corresponds to a planning unit, each column corresponds to a zone, and each cell indicates if the planning unit should be locked to a given zone. Thus each row should only contain at most a single `TRUE` value.

`character` field (column) name(s) that indicate if planning units should be locked for the solution. This type of argument is only compatible if the planning units in the argument to `x` are a `Spatial`, `sf::sf()`, or `data.frame` object. The fields (columns) must have `logical` (i.e. `TRUE` or `FALSE`) values indicating if the planning unit is to be locked for the solution. For problems containing multiple zones, this argument should contain a field (column) name for each management zone.

`Spatial` **or** `sf::sf()` planning units in `x` that spatially intersect with the argument to `y` (according to `intersecting_units()` are locked for to the solution. Note that this option is only available for problems that contain a single management zone.

`Raster` planning units in `x` that intersect with non-zero and non-`NA` raster cells are locked for the solution. For problems that contain multiple zones, the `Raster` object must contain a layer for each zone. Note that for multi-band arguments, each pixel must only contain a non-zero value in a single band. Additionally, if the cost data in `x` is a `Raster` object, we recommend standardizing `NA` values in this dataset with the cost data. In other words, the pixels in `x` that have `NA` values should also have `NA` values in the locked data.

See Also

constraints.

Examples

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

# load data
data(sim_pu_polygons, sim_features, sim_locked_in_raster)

# create minimal problem
p1 <- problem(sim_pu_polygons, sim_features, "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create problem with added locked in constraints using integers
p2 <- p1 %>% add_locked_in_constraints(which(sim_pu_polygons$locked_in))

# create problem with added locked in constraints using a field name
p3 <- p1 %>% add_locked_in_constraints("locked_in")

# create problem with added locked in constraints using raster data
p4 <- p1 %>% add_locked_in_constraints(sim_locked_in_raster)

# create problem with added locked in constraints using spatial polygon data
locked_in <- sim_pu_polygons[sim_pu_polygons$locked_in == 1, ]
p5 <- p1 %>% add_locked_in_constraints(locked_in)
## Not run:
# solve problems
s1 <- solve(p1)
s2 <- solve(p2)
s3 <- solve(p3)
s4 <- solve(p4)
s5 <- solve(p5)

# plot solutions
par(mfrow = c(3, 2), mar = c(0, 0, 4.1, 0))
plot(s1, main = "none locked in")
plot(s1[s1$solution_1 == 1, ], col = "darkgreen", add = TRUE)

plot(s2, main = "locked in (integer input)")
plot(s2[s2$solution_1 == 1, ], col = "darkgreen", add = TRUE)

plot(s3, main = "locked in (character input)")
plot(s3[s3$solution_1 == 1, ], col = "darkgreen", add = TRUE)

plot(s4, main = "locked in (raster input)")
plot(s4[s4$solution_1 == 1, ], col = "darkgreen", add = TRUE)
```

```
plot(s5, main = "locked in (polygon input)")
plot(s5[s5$solution_1 == 1, ], col = "darkgreen", add = TRUE)

# reset plot
par(mfrow = c(1, 1))

## End(Not run)

# create minimal multi-zone problem with spatial data
p6 <- problem(sim_pu_zones_polygons, sim_features_zones,
              cost_column = c("cost_1", "cost_2", "cost_3")) %>%
      add_min_set_objective() %>%
      add_absolute_targets(matrix(rpois(15, 1), nrow = 5,
                                  ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create multi-zone problem with locked in constraints using matrix data
locked_matrix <- sim_pu_zones_polygons@data[, c("locked_1", "locked_2",
                                                "locked_3")]
locked_matrix <- as.matrix(locked_matrix)

p7 <- p6 %>% add_locked_in_constraints(locked_matrix)
## Not run:
# solve problem
s6 <- solve(p6)

# create new column representing the zone id that each planning unit
# was allocated to in the solution
s6$solution <- category_vector(s6@data[, c("solution_1_zone_1",
                                           "solution_1_zone_2",
                                           "solution_1_zone_3")])
s6$solution <- factor(s6$solution)

# plot solution
spplot(s6, zcol = "solution", main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
# create multi-zone problem with locked in constraints using field names
p8 <- p6 %>% add_locked_in_constraints(c("locked_1", "locked_2", "locked_3"))
## Not run:
# solve problem
s8 <- solve(p8)

# create new column representing the zone id that each planning unit
# was allocated to in the solution
s8$solution <- category_vector(s8@data[, c("solution_1_zone_1",
                                           "solution_1_zone_2",
                                           "solution_1_zone_3")])
s8$solution[s8$solution == 1 & s8$solution_1_zone_1 == 0] <- 0
s8$solution <- factor(s8$solution)

# plot solution
```

```
spplot(s8, zcol = "solution", main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
# create multi-zone problem with raster planning units
p9 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_absolute_targets(matrix(rpois(15, 1), nrow = 5, ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create raster stack with locked in units
locked_in_stack <- sim_pu_zones_stack[[1]]
locked_in_stack[!is.na(locked_in_stack)] <- 0
locked_in_stack <- locked_in_stack[[c(1, 1, 1)]]
locked_in_stack[[1]][1] <- 1
locked_in_stack[[2]][2] <- 1
locked_in_stack[[3]][3] <- 1

# plot locked in stack
## Not run:
plot(locked_in_stack)

## End(Not run)
# add locked in raster units to problem
p9 <- p9 %>% add_locked_in_constraints(locked_in_stack)

## Not run:
# solve problem
s9 <- solve(p9)

# plot solution
plot(category_layer(s9), main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

`add_locked_out_constraints` *Add locked out constraints*

Description

Add constraints to a conservation planning `problem()` to ensure that specific planning units are not selected (or allocated to a specific zone) in the solution. For example, it may be useful to lock out planning units that have been degraded and are not suitable for conserving species. If specific planning units should be locked in to the solution, use `add_locked_out_constraints()`. For problems with non-binary planning unit allocations (e.g. proportions), the `add_manual_locked_constraints()` function can be used to lock planning unit allocations to a specific value.

Usage

```
add_locked_out_constraints(x, locked_out)

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

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

## S4 method for signature 'ConservationProblem,matrix'
add_locked_out_constraints(x, locked_out)

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

## S4 method for signature 'ConservationProblem,Spatial'
add_locked_out_constraints(x, locked_out)

## S4 method for signature 'ConservationProblem,sf'
add_locked_out_constraints(x, locked_out)

## S4 method for signature 'ConservationProblem,Raster'
add_locked_out_constraints(x, locked_out)
```

Arguments

`x`	`problem()` (i.e. `ConservationProblem`) object.
`locked_out`	Object that determines which planning units that should be locked out. See the Data format section for more information.

Value

Object (i.e. `ConservationProblem`) with the constraints added to it.

Data format

The locked planning units can be specified using the following formats. Generally, the locked data should correspond to the planning units in the argument to `x`. To help make working with `Raster` planning unit data easier, the locked data should correspond to cell indices in the `Raster` data. For example, `integer` arguments should correspond to cell indices and `logical` arguments should have a value for each cell—regardless of which planning unit cells contain `NA` values.

`integer vector` of indices pertaining to which planning units should be locked for the solution. This argument is only compatible with problems that contain a single zone.

`logical vector` containing `TRUE` and/or `FALSE` values that indicate which planning units should be locked in the solution. This argument is only compatible with problems that contain a single zone.

matrix containing logical TRUE and/or FALSE values which indicate if certain planning units are should be locked to a specific zone in the solution. Each row corresponds to a planning unit, each column corresponds to a zone, and each cell indicates if the planning unit should be locked to a given zone. Thus each row should only contain at most a single TRUE value.

character field (column) name(s) that indicate if planning units should be locked for the solution. This type of argument is only compatible if the planning units in the argument to x are a Spatial, sf::sf(), or data.frame object. The fields (columns) must have logical (i.e. TRUE or FALSE) values indicating if the planning unit is to be locked for the solution. For problems containing multiple zones, this argument should contain a field (column) name for each management zone.

Spatial **or** sf::sf() planning units in x that spatially intersect with the argument to y (according to intersecting_units() are locked for to the solution. Note that this option is only available for problems that contain a single management zone.

Raster planning units in x that intersect with non-zero and non-NA raster cells are locked for the solution. For problems that contain multiple zones, the Raster object must contain a layer for each zone. Note that for multi-band arguments, each pixel must only contain a non-zero value in a single band. Additionally, if the cost data in x is a Raster object, we recommend standardizing NA values in this dataset with the cost data. In other words, the pixels in x that have NA values should also have NA values in the locked data.

See Also

constraints.

Examples

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

# load data
data(sim_pu_polygons, sim_features, sim_locked_out_raster)

# create minimal problem
p1 <- problem(sim_pu_polygons, sim_features, "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create problem with added locked out constraints using integers
p2 <- p1 %>% add_locked_out_constraints(which(sim_pu_polygons$locked_out))

# create problem with added locked out constraints using a field name
p3 <- p1 %>% add_locked_out_constraints("locked_out")

# create problem with added locked out constraints using raster data
p4 <- p1 %>% add_locked_out_constraints(sim_locked_out_raster)

# create problem with added locked out constraints using spatial polygon data
locked_out <- sim_pu_polygons[sim_pu_polygons$locked_out == 1, ]
```

```
p5 <- p1 %>% add_locked_out_constraints(locked_out)
## Not run:
# solve problems
s1 <- solve(p1)
s2 <- solve(p2)
s3 <- solve(p3)
s4 <- solve(p4)
s5 <- solve(p5)

# plot solutions
par(mfrow = c(3,2), mar = c(0, 0, 4.1, 0))
plot(s1, main = "none locked out")
plot(s1[s1$solution_1 == 1, ], col = "darkgreen", add = TRUE)

plot(s2, main = "locked out (integer input)")
plot(s2[s2$solution_1 == 1, ], col = "darkgreen", add = TRUE)

plot(s3, main = "locked out (character input)")
plot(s3[s3$solution_1 == 1, ], col = "darkgreen", add = TRUE)

plot(s4, main = "locked out (raster input)")
plot(s4[s4$solution_1 == 1, ], col = "darkgreen", add = TRUE)

plot(s5, main = "locked out (polygon input)")
plot(s5[s5$solution_1 == 1, ], col = "darkgreen", add = TRUE)

# reset plot
par(mfrow = c(1, 1))

## End(Not run)

# create minimal multi-zone problem with spatial data
p6 <- problem(sim_pu_zones_polygons, sim_features_zones,
              cost_column = c("cost_1", "cost_2", "cost_3")) %>%
      add_min_set_objective() %>%
      add_absolute_targets(matrix(rpois(15, 1), nrow = 5, ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create multi-zone problem with locked out constraints using matrix data
locked_matrix <- sim_pu_zones_polygons@data[, c("locked_1", "locked_2",
                                                "locked_3")]
locked_matrix <- as.matrix(locked_matrix)

p7 <- p6 %>% add_locked_out_constraints(locked_matrix)
## Not run:
# solve problem
s6 <- solve(p6)

# create new column representing the zone id that each planning unit
# was allocated to in the solution
s6$solution <- category_vector(s6@data[, c("solution_1_zone_1",
                                           "solution_1_zone_2",
```

```
                                           "solution_1_zone_3")])
s6$solution <- factor(s6$solution)

# plot solution
spplot(s6, zcol = "solution", main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
# create multi-zone problem with locked out constraints using field names
p8 <- p6 %>% add_locked_out_constraints(c("locked_1", "locked_2",
                                          "locked_3"))
## Not run:
# solve problem
s8 <- solve(p8)

# create new column in s8 representing the zone id that each planning unit
# was allocated to in the solution
s8$solution <- category_vector(s8@data[, c("solution_1_zone_1",
                                           "solution_1_zone_2",
                                           "solution_1_zone_3")])
s8$solution[s8$solution == 1 & s8$solution_1_zone_1 == 0] <- 0
s8$solution <- factor(s8$solution)

# plot solution
spplot(s8, zcol = "solution", main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
# create multi-zone problem with raster planning units
p9 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_absolute_targets(matrix(rpois(15, 1), nrow = 5, ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create raster stack with locked out units
locked_out_stack <- sim_pu_zones_stack[[1]]
locked_out_stack[!is.na(locked_out_stack)] <- 0
locked_out_stack <- locked_out_stack[[c(1, 1, 1)]]
locked_out_stack[[1]][1] <- 1
locked_out_stack[[2]][2] <- 1
locked_out_stack[[3]][3] <- 1

# plot locked out stack
## Not run:
plot(locked_out_stack)

## End(Not run)
# add locked out raster units to problem
p9 <- p9 %>% add_locked_out_constraints(locked_out_stack)

## Not run:
# solve problem
s9 <- solve(p9)
```

```
# plot solution
plot(category_layer(s9), main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

---

add_loglinear_targets *Add targets using log-linear scaling*

Description

Add targets to a conservation planning `problem()` by log-linearly interpolating the targets between thresholds based on the total amount of each feature in the study area (Rodrigues *et al.* 2004). Additionally, caps can be applied to targets to prevent features with massive distributions from being over-represented in solutions (Butchart *et al.* 2015).

Usage

```
add_loglinear_targets(
  x,
  lower_bound_amount,
  lower_bound_target,
  upper_bound_amount,
  upper_bound_target,
  cap_amount = NULL,
  cap_target = NULL,
  abundances = feature_abundances(x, na.rm = FALSE)$absolute_abundance
)
```

Arguments

`x` `problem()` (i.e. `ConservationProblem`) object.

`lower_bound_amount` `numeric` threshold.

`lower_bound_target` `numeric` relative target that should be applied to features with a total amount that is less than or equal to `lower_bound_amount`.

`upper_bound_amount` `numeric` threshold.

`upper_bound_target` `numeric` relative target that should be applied to features with a total amount that is greater than or equal to `upper_bound_amount`.

`cap_amount` `numeric` total amount at which targets should be capped. Defaults to `NULL` so that targets are not capped.

`cap_target` `numeric` amount-based target to apply to features which have a total amount greater than argument to `cap_amount`. Defaults to `NULL` so that targets are not capped.

abundances — numeric total amount of each feature to use when calculating the targets. Defaults to the feature abundances in the study area (calculated using the feature_abundances() function.

Details

Targets are used to specify the minimum amount or proportion of a feature's distribution that needs to be protected. All conservation planning problems require adding targets with the exception of the maximum cover problem (see add_max_cover_objective()), which maximizes all features in the solution and therefore does not require targets.

Seven parameters are used to calculate the targets: lower_bound_amount specifies the first range size threshold, lower_bound_target specifies the relative target required for species with a range size equal to or less than the first threshold, upper_bound_amount specifies the second range size threshold, upper_bound_target specifies the relative target required for species with a range size equal to or greater than the second threshold, cap_amount specifies the third range size threshold, cap_target specifies the absolute target that is uniformly applied to species with a range size larger than that third threshold, and finally abundances specifies the range size for each feature that should be used when calculating the targets.

The target calculations do not account for the size of each planning unit. Therefore, the feature data should account for the size of each planning unit if this is important (e.g. pixel values in the argument to features in the function problem() could correspond to amount of land occupied by the feature in km^2 units). Additionally, the function can only be applied to ConservationProblem objects that are associated with a single zone.

Value

Object (i.e. ConservationProblem) with the targets added to it.

Notes

Early versions (< 5.0.2.4) used different equations for calculating targets.

References

Rodrigues ASL, Akcakaya HR, Andelman SJ, Bakarr MI, Boitani L, Brooks TM, Chanson JS, Fishpool LDC, da Fonseca GAB, Gaston KJ, and others (2004) Global gap analysis: priority regions for expanding the global protected-area network. *BioScience*, 54: 1092–1100.

Butchart SHM, Clarke M, Smith RJ, Sykes RE, Scharlemann JPW, Harfoot M, Buchanan, GM, Angulo A, Balmford A, Bertzky B, and others (2015) Shortfalls and solutions for meeting national and global conservation area targets. *Conservation Letters*, 8: 329–337.

See Also

targets, loglinear_interpolation().

Examples

```
# load data
data(sim_pu_raster, sim_features)

# create problem using loglinear targets
p <- problem(sim_pu_raster, sim_features) %>%
     add_min_set_objective() %>%
     add_loglinear_targets(10, 0.9, 100, 0.2) %>%
     add_binary_decisions() %>%
     add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s <- solve(p)

# plot solution
plot(s, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

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

Description

Specify that the *SYMPHONY* software (Ralphs & Güzelsoy 2005) – using the **lpsymphony** package – should be used to solve a conservation planning `problem()`. This function can also be used to customize the behavior of the solver. It requires the **lpsymphony** package to be installed (see below for installation instructions).

Usage

```
add_lpsymphony_solver(
  x,
  gap = 0.1,
  time_limit = .Machine$integer.max,
  first_feasible = FALSE,
  verbose = TRUE
)
```

Arguments

`x`	`problem()` (i.e. `ConservationProblem`) 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 0.1 (i.e. 10% from optimality).

time_limit
: `numeric` time limit (seconds) for generating solutions. The solver will return the current best solution when this time limit is exceeded. The default value is the largest integer value (i.e. `.Machine$integer.max`), effectively meaning that solver will keep running until a solution within the optimality gap is found.

first_feasible
: `logical` should the first feasible solution be be returned? If `first_feasible` is set to `TRUE`, the solver will return the first solution it encounters that meets all the constraints, regardless of solution quality. Note that the first feasible solution is not an arbitrary solution, rather it is derived from the relaxed solution, and is therefore often reasonably close to optimality. Defaults to `FALSE`.

verbose
: `logical` should information be printed while solving optimization problems? Defaults to `TRUE`.

Details

SYMPHONY is an open-source mixed integer programming solver that is part of the Computational Infrastructure for Operations Research (COIN-OR) project. This solver is provided because it may be easier to install on some systems than the **Rsymphony** package. Additionally – although the **lpsymphony** package doesn't provide the functionality to specify the number of threads for solving a problem – the **lpsymphony** package will solve problems using parallel processing (unlike the **Rsymphony** package). As a consequence, this solver will likely generate solutions much faster than the `add_rsymphony_solver()`. Although formal benchmarks examining the performance of this solver have yet to be completed, please see Schuster *et al.* (2020) for benchmarks comparing the run time and solution quality of the **Rsymphony** solver.

Value

Object (i.e. `ConservationProblem`) with the solver added to it.

Installation

The **lpsymphony** package is distributed through Bioconductor. To install the **lpsymphony** package, please use the following code:

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

References

Ralphs TK and Güzelsoy M (2005) The SYMPHONY callable library for mixed integer programming. In The Next Wave in Computing, Optimization, and Decision Technologies (pp. 61–76). Springer, Boston, MA.

Schuster R, Hanson JO, Strimas-Mackey M, and Bennett JR (2020). Exact integer linear programming solvers outperform simulated annealing for solving conservation planning problems. *PeerJ*, 8: e9258.

See Also

solvers.

Examples

```
## Not run:
# load data
data(sim_pu_raster, sim_features)

# create problem
p <- problem(sim_pu_raster, sim_features) %>%
  add_min_set_objective() %>%
  add_relative_targets(0.05) %>%
  add_proportion_decisions() %>%
  add_lpsymphony_solver(time_limit = 5, verbose = FALSE)

# generate solution
s <- solve(p)

# plot solution
plot(s, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

`add_mandatory_allocation_constraints`
Add mandatory allocation constraints

Description

Add constraints to ensure that every planning unit is allocated to a management zone in the solution. Note that this function can only be used with problems that contain multiple zones.

Usage

```
## S4 method for signature 'ConservationProblem'
add_mandatory_allocation_constraints(x)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object.

Details

For a conservation planning `problem()` with multiple management zones, it may sometimes be desirable to obtain a solution that assigns each and every single planning unit to a zone. For example, when developing land-use plans, some decision makers may require that each and every single parcel of land has been allocated a specific land-use type. In other words are no "left over" areas. Although it might seem tempting to simply solve the problem and manually assign "left over" planning units to a default zone afterwards (e.g. an "other", "urban", or "grazing" land-use), this could result in highly sub-optimal solutions if there penalties for siting the default land-use adjacent to other zones. Instead, this function can be used to specify that all planning units in a problem with multiple zones must be allocated to a management zone (i.e. zone allocation is mandatory).

Value

Object (i.e. ConservationProblem) with the constraints added to it.

See Also

constraints.

Examples

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

# load data
data(sim_pu_zones_stack, sim_features_zones)

# create multi-zone problem with minimum set objective
targets_matrix <- matrix(rpois(15, 1), nrow = 5, ncol = 3)

# create minimal problem with minimum set objective
p1 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_absolute_targets(targets_matrix) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create another problem that is the same as p1, but has constraints
# to mandate that every planning unit in the solution is assigned to
# zone
p2 <- p1 %>% add_mandatory_allocation_constraints()
## Not run:
# solve problems
s1 <- solve(p1)
s2 <- solve(p2)

# convert solutions into category layers, where each pixel is assigned
 # value indicating which zone it was assigned to in the zone
c1 <- category_layer(s1)
c2 <- category_layer(s2)

# plot solution category layers
plot(stack(c1, c2), main = c("default", "mandatory allocation"),
     axes = FALSE, box = FALSE)

## End(Not run)
```

---

add_manual_bounded_constraints

Add manually specified bounds constraints

Description

Add constraints to a conservation planning `problem()` to ensure that the planning unit values (e.g. proportion, binary) in a solution range between specific lower and upper bounds. This function offers more fine-grained control than the `add_manual_locked_constraints()` function and is is most useful for problems involving proportion-type or semi-continuous decisions.

Usage

```
add_manual_bounded_constraints(x, data)

## S4 method for signature 'ConservationProblem,data.frame'
add_manual_bounded_constraints(x, data)

## S4 method for signature 'ConservationProblem,tbl_df'
add_manual_bounded_constraints(x, data)
```

Arguments

`x`	`problem()` (i.e. `ConservationProblem`) object.
`data`	`data.frame` or `tibble::tibble()` object. See the Data format section for more information.

Value

Object (i.e. `ConservationProblem`) with the constraints added to it.

Data format

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

pu `integer` planning unit identifier.

zone `character` names of zones. Note that this argument is optional for arguments to `x` that contain a single zone.

lower `numeric` values indicating the minimum value that each planning unit can be allocated to in each zone in the solution.

upper `numeric` values indicating the maximum value that each planning unit can be allocated to in each zone in the solution.

See Also

constraints.

Examples

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

# load data
data(sim_pu_polygons, sim_features, sim_pu_zones_polygons,
```

```
     sim_features_zones)

# create minimal problem
p1 <- problem(sim_pu_polygons, sim_features, "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create problem with locked in constraints using add_locked_constraints
p2 <- p1 %>% add_locked_in_constraints("locked_in")

# create identical problem using add_manual_bounded_constraints
bounds_data <- data.frame(pu = which(sim_pu_polygons$locked_in),
                          lower = 1, upper = 1)

p3 <- p1 %>% add_manual_bounded_constraints(bounds_data)
## Not run:
# solve problems
s1 <- solve(p1)
s2 <- solve(p2)
s3 <- solve(p3)

# plot solutions
par(mfrow = c(1,3), mar = c(0, 0, 4.1, 0))
plot(s1, main = "none locked in")
plot(s1[s1$solution_1 == 1, ], col = "darkgreen", add = TRUE)

plot(s2, main = "add_locked_in_constraints")
plot(s2[s2$solution_1 == 1, ], col = "darkgreen", add = TRUE)

plot(s3, main = "add_bounds_constraints")
plot(s3[s3$solution_1 == 1, ], col = "darkgreen", add = TRUE)

## End(Not run)
# create minimal problem with multiple zones
p4 <- problem(sim_pu_zones_polygons, sim_features_zones,
              c("cost_1", "cost_2", "cost_3")) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create data.frame with the following constraints:
# planning units 1, 2, and 3 must be allocated to zone 1 in the solution
# planning units 4, and 5 must be allocated to zone 2 in the solution
# planning units 8 and 9 must not be allocated to zone 3 in the solution
bounds_data2 <- data.frame(pu = c(1, 2, 3, 4, 5, 8, 9),
                           zone = c(rep("zone_1", 3), rep("zone_2", 2),
                                    rep("zone_3", 2)),
                           lower = c(rep(1, 5), rep(0, 2)),
                           upper = c(rep(1, 5), rep(0, 2)))
```

```
# print bounds data
print(bounds_data2)

# create problem with added constraints
p5 <- p4 %>% add_manual_bounded_constraints(bounds_data2)
## Not run:
# solve problem
s4 <- solve(p4)
s5 <- solve(p5)

# create two new columns representing the zone id that each planning unit
# was allocated to in the two solutions
s4$solution <- category_vector(s4@data[, c("solution_1_zone_1",
                                           "solution_1_zone_2",
                                           "solution_1_zone_3")])
s4$solution <- factor(s4$solution)

s4$solution_bounded <- category_vector(s5@data[, c("solution_1_zone_1",
                                                   "solution_1_zone_2",
                                                   "solution_1_zone_3")])
s4$solution_bounded <- factor(s4$solution_bounded)

# plot solutions
spplot(s4, zcol = c("solution", "solution_bounded"), axes = FALSE,
       box = FALSE)

## End(Not run)
```

add_manual_locked_constraints

Add manually specified locked constraints

Description

Add constraints to a conservation planning problem() to ensure that solutions allocate (or do not allocate) specific planning units to specific management zones. This function offers more fine-grained control than the add_locked_in_constraints() and add_locked_out_constraints() functions.

Usage

```
add_manual_locked_constraints(x, data)

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

## S4 method for signature 'ConservationProblem,tbl_df'
add_manual_locked_constraints(x, data)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object.
data	`data.frame` or `tibble::tibble()` object. See the Data format section for more information.

Value

Object (i.e. `ConservationProblem`) with the constraints added to it.

Data format

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

pu `integer` planning unit identifier.

zone `character` names of zones. Note that this argument is optional for arguments to `x` that contain a single zone.

status `numeric` values indicating how much of each planning unit should be allocated to each zone in the solution. For example, the `numeric` values could be binary values (i.e. zero or one) for problems containing binary-type decision variables (using the `add_binary_decisions()` function). Alternatively, the `numeric` values could be proportions (e.g. 0.5) for problems containing proportion-type decision variables (using the `add_proportion_decisions()`).

See Also

constraints.

Examples

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

# load data
data(sim_pu_polygons, sim_features, sim_pu_zones_polygons,
     sim_features_zones)

# create minimal problem
p1 <- problem(sim_pu_polygons, sim_features, "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create problem with locked in constraints using add_locked_constraints
p2 <- p1 %>% add_locked_in_constraints("locked_in")

# create identical problem using add_manual_locked_constraints
locked_data <- data.frame(pu = which(sim_pu_polygons$locked_in),
                          status = 1)

p3 <- p1 %>% add_manual_locked_constraints(locked_data)
```

```
## Not run:
# solve problems
s1 <- solve(p1)
s2 <- solve(p2)
s3 <- solve(p3)

# plot solutions
par(mfrow = c(1,3), mar = c(0, 0, 4.1, 0))
plot(s1, main = "none locked in")
plot(s1[s1$solution_1 == 1, ], col = "darkgreen", add = TRUE)

plot(s2, main = "add_locked_in_constraints")
plot(s2[s2$solution_1 == 1, ], col = "darkgreen", add = TRUE)

plot(s3, main = "add_manual_constraints")
plot(s3[s3$solution_1 == 1, ], col = "darkgreen", add = TRUE)

## End(Not run)
# create minimal problem with multiple zones
p4 <- problem(sim_pu_zones_polygons, sim_features_zones,
              c("cost_1", "cost_2", "cost_3")) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create data.frame with the following constraints:
# planning units 1, 2, and 3 must be allocated to zone 1 in the solution
# planning units 4, and 5 must be allocated to zone 2 in the solution
# planning units 8 and 9 must not be allocated to zone 3 in the solution
locked_data2 <- data.frame(pu = c(1, 2, 3, 4, 5, 8, 9),
                           zone = c(rep("zone_1", 3), rep("zone_2", 2),
                                    rep("zone_3", 2)),
                           status = c(rep(1, 5), rep(0, 2)))

# print locked constraint data
print(locked_data2)

# create problem with added constraints
p5 <- p4 %>% add_manual_locked_constraints(locked_data2)
## Not run:
# solve problem
s4 <- solve(p4)
s5 <- solve(p5)

# create two new columns representing the zone id that each planning unit
# was allocated to in the two solutions
s4$solution <- category_vector(s4@data[, c("solution_1_zone_1",
                                           "solution_1_zone_2",
                                           "solution_1_zone_3")])
s4$solution <- factor(s4$solution)
```

```
s4$solution_locked <- category_vector(s5@data[, c("solution_1_zone_1",
                                                   "solution_1_zone_2",
                                                   "solution_1_zone_3")])
s4$solution_locked <- factor(s4$solution_locked)

# plot solutions
spplot(s4, zcol = c("solution", "solution_locked"), axes = FALSE,
       box = FALSE)

## End(Not run)
```

---

add_manual_targets *Add manual targets*

Description

Set targets for a conservation planning problem() by manually specifying all the required information for each target. This function is useful because it can be used to customize all aspects of a target. For most cases, targets can be specified using the add_absolute_targets() and add_relative_targets() functions. However, this function can be used to (i) mix absolute and relative targets for different features and zones, (ii) set targets that pertain to the allocations of planning units in multiple zones, and (iii) set targets that require different senses (e.g. targets which specify the solution should not exceed a certain quantity using "<=" values).

Usage

```
add_manual_targets(x, targets)

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

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

Arguments

x	problem() (i.e. ConservationProblem) object.
targets	data.frame or tibble::tibble() object. See the Target data format section for more information.

Details

Targets are used to specify the minimum amount or proportion of a feature's distribution that needs to be protected. Most conservation planning problems require targets with the exception of the maximum cover (see add_max_cover_objective()) and maximum utility (see add_max_utility_objective()) problems. Attempting to solve problems with objectives that require targets without specifying targets will throw an error.

For problems associated with multiple management zones, this function can be used to set targets that each pertain to a single feature and a single zone. To set targets which can be met through allocating different planning units to multiple zones, see the add_manual_targets() function. An example of a target that could be met through allocations to multiple zones might be where each management zone is expected to result in a different amount of a feature and the target requires that the total amount of the feature in all zones must exceed a certain threshold. In other words, the target does not require that any single zone secure a specific amount of the feature, but the total amount held in all zones must secure a specific amount. Thus the target could, potentially, be met through allocating all planning units to any specific management zone, or through allocating the planning units to different combinations of management zones.

Value

Object (i.e. ConservationProblem) with the targets added to it.

Target data format

The targets argument should contain the following fields (columns):

feature character name of features in argument to x.

zone character name of zones in argument to x. This field (column) is optional for arguments to x that do not contain multiple zones.

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

sense character sense of the target. Acceptable values include: ">=", "<=", and "=". This field (column) is optional and if it is missing then target senses will default to ">=" values.

target numeric target threshold.

See Also

targets.

Examples

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

# load data
data(sim_pu_raster, sim_features, sim_pu_zones_stack, sim_features_zones)

# create problem with 10% relative targets
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s1 <- solve(p1)
```

```
# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
# create equivalent problem using add_manual_targets
p2 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_manual_targets(data.frame(feature = names(sim_features),
                                    type = "relative", sense = ">=",
                                    target = 0.1)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s2 <- solve(p2)

# plot solution
plot(s2, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
# create problem with targets set for only a few features
p3 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_manual_targets(data.frame(
        feature = names(sim_features)[1:3], type = "relative",
        sense = ">=", target = 0.1)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s3 <- solve(p3)

# plot solution
plot(s3, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
# create problem that aims to secure at least 10% of the habitat for one
# feature whilst ensuring that the solution does not capture more than
# 20 units habitat for different feature
# create problem with targets set for only a few features
p4 <- problem(sim_pu_raster, sim_features[[1:2]]) %>%
      add_min_set_objective() %>%
      add_manual_targets(data.frame(
        feature = names(sim_features)[1:2], type = "relative",
        sense = c(">=", "<="), target = c(0.1, 0.2))) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s4 <- solve(p4)

# plot solution
```

```
plot(s4, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
# create a multi-zone problem that requires a specific amount of each
# feature in each zone
targets_matrix <- matrix(rpois(15, 1), nrow = 5, ncol = 3)

p5 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_absolute_targets(targets_matrix) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s5 <- solve(p5)

# plot solution
plot(category_layer(s5), main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
# create equivalent problem using add_manual_targets
targets_dataframe <- expand.grid(feature = feature_names(sim_features_zones),
                                 zone = zone_names(sim_features_zones),
                                 sense = ">=", type = "absolute")
targets_dataframe$target <- c(targets_matrix)

p6 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_manual_targets(targets_dataframe) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s6 <- solve(p6)

# plot solution
plot(category_layer(s6), main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
# create a problem that requires a total of 20 units of habitat to be
# captured for two species. This can be achieved through representing
# habitat in two zones. The first zone represents a full restoration of the
# habitat and a second zone represents a partial restoration of the habitat
# Thus only half of the benefit that would have been gained from the full
# restoration is obtained when planning units are allocated a partial
# restoration

# create data
spp_zone1 <- as.list(sim_features_zones)[[1]][[1:2]]
spp_zone2 <- spp_zone1 * 0.5
costs <- sim_pu_zones_stack[[1:2]]

# create targets
```

```
targets_dataframe2 <- tibble::tibble(
  feature = names(spp_zone1), zone = list(c("z1", "z2"), c("z1", "z2")),
  sense = c(">=", ">="), type = c("absolute", "absolute"),
  target = c(20, 20))

# create problem
p7 <- problem(costs, zones(spp_zone1, spp_zone2,
                           feature_names = names(spp_zone1),
                           zone_names = c("z1", "z2"))) %>%
      add_min_set_objective() %>%
      add_manual_targets(targets_dataframe2) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s7 <- solve(p7)

# plot solution
plot(category_layer(s7), main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

`add_max_cover_objective` *Add maximum coverage objective*

Description

Set the objective of a conservation planning `problem()` to represent at least one instance of as many features as possible within a given budget. This type of objective does not use targets, and feature weights should be used instead to increase the representation of different features in solutions.

Usage

```
add_max_cover_objective(x, budget)
```

Arguments

`x`	`problem()` (i.e. `ConservationProblem`) object.
`budget`	`numeric` value specifying the maximum expenditure of the prioritization. For problems with multiple zones, the argument to `budget` can be a single `numeric` value to specify a budget for the entire solution or a `numeric` `vector` to specify a budget for each each management zone.

Details

A problem objective is used to specify the overall goal of the conservation planning problem. Please note that all conservation planning problems formulated in the **prioritizr** package require the addition of objectives—failing to do so will return an error message when attempting to solve problem.

The maximum coverage objective seeks to find the set of planning units that maximizes the number of represented features, while keeping cost within a fixed budget. Here, features are treated as being represented if the reserve system contains at least a single instance of a feature (i.e. an amount greater than 1). This formulation has often been used in conservation planning problems dealing with binary biodiversity data that indicate the presence/absence of suitable habitat (e.g. Church & Velle 1974). Additionally, weights can be used to favor the representation of certain features over other features (see `add_feature_weights()`). Check out the `add_max_features_objective()` for a more generalized formulation which can accommodate user-specified representation targets.

This formulation is based on the historical maximum coverage reserve selection formulation (Church & Velle 1974; Church *et al.* 1996). The maximum coverage objective for the reserve design problem can be expressed mathematically for a set of planning units (I indexed by i) and a set of features (J indexed by j) as:

$$Maximize \sum_{i=1}^{I} -sc_i x_i + \sum_{j=1}^{J} y_j w_j subjectto \sum_{i=1}^{I} x_i r_{ij} \geq y_j \times 1 \forall j \in J \sum_{i=1}^{I} x_i c_i \leq B$$

Here, x_i is the decisions variable (e.g. specifying whether planning unit i has been selected (1) or not (0)), r_{ij} is the amount of feature j in planning unit i, y_j indicates if the solution has meet the target t_j for feature j, and w_j is the weight for feature j (defaults to 1 for all features; see `add_feature_weights()` to specify weights). Additionally, B is the budget allocated for the solution, c_i is the cost of planning unit i, and s is a scaling factor used to shrink the costs so that the problem will return a cheapest solution when there are multiple solutions that represent the same amount of all features within the budget.

Value

Object (i.e. `ConservationProblem`) with the objective added to it.

Notes

In early versions (< 3.0.0.0), the mathematical formulation underpinning this function was very different. Specifically, as described above, the function now follows the formulations outlined in Church *et al.* (1996). The old formulation is now provided by the `add_max_utility_objective()` function.

References

Church RL and Velle CR (1974) The maximum covering location problem. *Regional Science*, 32: 101–118.

Church RL, Stoms DM, and Davis FW (1996) Reserve selection as a maximum covering location problem. *Biological Conservation*, 76: 105–112.

See Also

`add_feature_weights()`, objectives.

Examples

```
# load data
data(sim_pu_raster, sim_pu_zones_stack, sim_features, sim_features_zones)

# threshold the feature data to generate binary biodiversity data
sim_binary_features <- sim_features
thresholds <- raster::quantile(sim_features, probs = 0.95, names = FALSE,
                               na.rm = TRUE)
for (i in seq_len(raster::nlayers(sim_features)))
  sim_binary_features[[i]] <- as.numeric(raster::values(sim_features[[i]]) >
                                         thresholds[[i]])

# create problem with maximum utility objective
p1 <- problem(sim_pu_raster, sim_binary_features) %>%
      add_max_cover_objective(500) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s1 <- solve(p1)

# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)

# threshold the multi-zone feature data to generate binary biodiversity data
sim_binary_features_zones <- sim_features_zones
for (z in number_of_zones(sim_features_zones)) {
  thresholds <- raster::quantile(sim_features_zones[[z]], probs = 0.95,
                                 names = FALSE, na.rm = TRUE)
  for (i in seq_len(number_of_features(sim_features_zones))) {
    sim_binary_features_zones[[z]][[i]] <- as.numeric(
      raster::values(sim_features_zones[[z]][[i]]) > thresholds[[i]])
  }
}

# create multi-zone problem with maximum utility objective that
# has a single budget for all zones
p2 <- problem(sim_pu_zones_stack, sim_binary_features_zones) %>%
      add_max_cover_objective(800) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s2 <- solve(p2)

# plot solution
plot(category_layer(s2), main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

```
# create multi-zone problem with maximum utility objective that
# has separate budgets for each zone
p3 <- problem(sim_pu_zones_stack, sim_binary_features_zones) %>%
      add_max_cover_objective(c(400, 400, 400)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s3 <- solve(p3)

# plot solution
plot(category_layer(s3), main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

---

add_max_features_objective

Add maximum feature representation objective

Description

Set the objective of a conservation planning problem() to fulfill as many targets as possible while ensuring that the cost of the solution does not exceed a budget.

Usage

```
add_max_features_objective(x, budget)
```

Arguments

x	problem() (i.e. ConservationProblem) object.
budget	numeric value specifying the maximum expenditure of the prioritization. For problems with multiple zones, the argument to budget can be a single numeric value to specify a budget for the entire solution or a numeric vector to specify a budget for each each management zone.

Details

A problem objective is used to specify the overall goal of the conservation planning problem. Please note that all conservation planning problems formulated in the **prioritizr** package require the addition of objectives—failing to do so will return an error message when attempting to solve problem.

The maximum feature representation objective is an enhanced version of the maximum coverage objective add_max_cover_objective() because targets can be used to ensure that a certain amount of each feature is required in order for them to be adequately represented (similar to the minimum set objective (see add_min_set_objective()). This objective finds the set of planning units that meets representation targets for as many features as possible while staying within a fixed budget (inspired by Cabeza and Moilanen 2001). Additionally, weights can be used add_feature_weights()). If multiple solutions can meet the same number of weighted targets while staying within budget, the cheapest solution is returned.

The maximum feature objective for the reserve design problem can be expressed mathematically for a set of planning units (I indexed by i) and a set of features (J indexed by j) as:

$$Maximize \sum_{i=1}^{I} -sc_i x_i + \sum_{j=1}^{J} y_j w_j subjectto \sum_{i=1}^{I} x_i r_{ij} \geq y_j t_j \forall j \in J \sum_{i=1}^{I} x_i c_i \leq B$$

Here, x_i is the decisions variable (e.g. specifying whether planning unit i has been selected (1) or not (0)), r_{ij} is the amount of feature j in planning unit i, t_j is the representation target for feature j, y_j indicates if the solution has meet the target t_j for feature j, and w_j is the weight for feature j (defaults to 1 for all features; see `add_feature_weights()` to specify weights). Additionally, B is the budget allocated for the solution, c_i is the cost of planning unit i, and s is a scaling factor used to shrink the costs so that the problem will return a cheapest solution when there are multiple solutions that represent the same amount of all features within the budget.

Value

Object (i.e. `ConservationProblem`) with the objective added to it.

References

Cabeza M and Moilanen A (2001) Design of reserve networks and the persistence of biodiversity. *Trends in Ecology & Evolution*, 16: 242–248.

See Also

`add_feature_weights()`, objectives.

Examples

```
# load data
data(sim_pu_raster, sim_pu_zones_stack, sim_features, sim_features_zones)

# create problem with maximum features objective
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_max_features_objective(1800) %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s1 <- solve(p1)

# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)

# create multi-zone problem with maximum features objective,
```

```
# with 10% representation targets for each feature, and set
# a budget such that the total maximum expenditure in all zones
# cannot exceed 3000
p2 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_max_features_objective(3000) %>%
      add_relative_targets(matrix(0.1, ncol = 3, nrow = 5)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s2 <- solve(p2)

# plot solution
plot(category_layer(s2), main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
# create multi-zone problem with maximum features objective,
# with 10% representation targets for each feature, and set
# separate budgets for each management zone
p3 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_max_features_objective(c(3000, 3000, 3000)) %>%
      add_relative_targets(matrix(0.1, ncol = 3, nrow = 5)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s3 <- solve(p3)

# plot solution
plot(category_layer(s3), main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

`add_max_phylo_div_objective`

Add maximum phylogenetic diversity objective

Description

Set the objective of a conservation planning `problem()` to maximize the phylogenetic diversity of the features represented in the solution subject to a budget. This objective is similar to `add_max_features_objective()` except that emphasis is placed on representing a phylogenetically diverse set of species, rather than as many features as possible (subject to weights). This function was inspired by Faith (1992) and Rodrigues *et al.* (2002).

Usage

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

Arguments

x	problem() (i.e. ConservationProblem) object.
budget	numeric value specifying the maximum expenditure of the prioritization. For problems with multiple zones, the argument to budget can be a single numeric value to specify a budget for the entire solution or a numeric vector to specify a budget for each each management zone.
tree	phylo() object specifying a phylogenetic tree for the conservation features.

Details

A problem objective is used to specify the overall goal of the conservation planning problem. Please note that all conservation planning problems formulated in the **prioritizr** package require the addition of objectives—failing to do so will return an error message when attempting to solve problem.

The maximum phylogenetic diversity objective finds the set of planning units that meets representation targets for a phylogenetic tree while staying within a fixed budget. If multiple solutions can meet all targets while staying within budget, the cheapest solution is chosen. Note that this objective is similar to the maximum features objective (add_max_features_objective()) in that it allows for both a budget and targets to be set for each feature. However, unlike the maximum feature objective, the aim of this objective is to maximize the total phylogenetic diversity of the targets met in the solution, so if multiple targets are provided for a single feature, the problem will only need to meet a single target for that feature for the phylogenetic benefit for that feature to be counted when calculating the phylogenetic diversity of the solution. In other words, for multi-zone problems, this objective does not aim to maximize the phylogenetic diversity in each zone, but rather this objective aims to maximize the phylogenetic diversity of targets that can be met through allocating planning units to any of the different zones in a problem. This can be useful for problems where targets pertain to the total amount held for each feature across multiple zones. For example, each feature might have a non-zero amount of suitable habitat in each planning unit when the planning units are assigned to a (i) not restored, (ii) partially restored, or (iii) completely restored management zone. Here each target corresponds to a single feature and can be met through the total amount of habitat in planning units present to the three zones.

The maximum phylogenetic diversity objective for the reserve design problem can be expressed mathematically for a set of planning units (I indexed by i) and a set of features (J indexed by j) as:

$$Maximize \sum_{i=1}^{I} -sc_i x_i + \sum_{j=1}^{J} m_b l_b subjectto \sum_{i=1}^{I} x_i r_{ij} \geq y_j t_j \forall j \in J m_b \leq y_j \forall j \in T(b) \sum_{i=1}^{I} x_i c_i \leq B$$

Here, x_i is the decisions variable (e.g. specifying whether planning unit i has been selected (1) or not (0)), r_{ij} is the amount of feature j in planning unit i, t_j is the representation target for feature j, y_j indicates if the solution has meet the target t_j for feature j. Additionally, T represents a phylogenetic tree containing features j and has the branches b associated within lengths l_b. The binary variable m_b denotes if at least one feature associated with the branch b has met its representation as indicated by y_j. For brevity, we denote the features j associated with branch b using $T(b)$. Finally, B is the budget allocated for the solution, c_i is the cost of planning unit i, and s is a scaling factor used to shrink the costs so that the problem will return a cheapest solution when there are multiple solutions that represent the same amount of all features within the budget.

Value

Object (i.e. ConservationProblem) with the objective added to it.

Notes

In early versions, this function was named as the `add_max_phylo_div_objective` function.

References

Faith DP (1992) Conservation evaluation and phylogenetic diversity. *Biological Conservation*, 61: 1–10.

Rodrigues ASL and Gaston KJ (2002) Maximising phylogenetic diversity in the selection of networks of conservation areas. *Biological Conservation*, 105: 103–111.

See Also

objectives, `branch_matrix()`.

Examples

```
# load ape package
require(ape)

# load data
data(sim_pu_raster, sim_features, sim_phylogeny, sim_pu_zones_stack,
     sim_features_zones)

# plot the simulated phylogeny
## Not run:
par(mfrow = c(1, 1))
plot(sim_phylogeny, main = "phylogeny")

## End(Not run)
# create problem with a maximum phylogenetic diversity objective,
# where each feature needs 10% of its distribution to be secured for
# it to be adequately conserved and a total budget of 1900
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_max_phylo_div_objective(1900, sim_phylogeny) %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s1 <- solve(p1)

# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)

# find out which features have their targets met
r1 <- eval_target_coverage_summary(p1, s1)
print(r1, width = Inf)
```

```
# plot the phylogeny and color the adequately represented features in red
plot(sim_phylogeny, main = "adequately represented features",
     tip.color = replace(
       rep("black", nlayers(sim_features)),
       sim_phylogeny$tip.label %in% r1$feature[r1$met],
       "red"))

## End(Not run)
# rename the features in the example phylogeny for use with the
# multi-zone data
sim_phylogeny$tip.label <- feature_names(sim_features_zones)

# create targets for a multi-zone problem. Here, each feature needs a total
# of 10 units of habitat to be conserved among the three zones to be
# considered adequately conserved
targets <- tibble::tibble(
  feature = feature_names(sim_features_zones),
  zone = list(zone_names(sim_features_zones))[rep(1,
         number_of_features(sim_features_zones))],
  type = rep("absolute", number_of_features(sim_features_zones)),
  target = rep(10, number_of_features(sim_features_zones)))

# create a multi-zone problem with a maximum phylogenetic diversity
# objective, where the total expenditure in all zones is 5000.
p2 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_max_phylo_div_objective(5000, sim_phylogeny) %>%
      add_manual_targets(targets) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s2 <- solve(p2)

# plot solution
plot(category_layer(s2), main = "solution", axes = FALSE, box = FALSE)

# find out which features have their targets met
r2 <- eval_target_coverage_summary(p2, s2)
print(r2, width = Inf)

# plot the phylogeny and color the adequately represented features in red
plot(sim_phylogeny, main = "adequately represented features",
     tip.color = replace(rep("black", nlayers(sim_features)),
                         which(r2$met), "red"))

## End(Not run)
# create a multi-zone problem with a maximum phylogenetic diversity
# objective, where each zone has a separate budget.
p3 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_max_phylo_div_objective(c(2500, 500, 2000), sim_phylogeny) %>%
      add_manual_targets(targets) %>%
      add_binary_decisions() %>%
```

```
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s3 <- solve(p3)

# plot solution
plot(category_layer(s3), main = "solution", axes = FALSE, box = FALSE)

# find out which features have their targets met
r3 <- eval_target_coverage_summary(p3, s3)
print(r3, width = Inf)

# plot the phylogeny and color the adequately represented features in red
plot(sim_phylogeny, main = "adequately represented features",
     tip.color = replace(rep("black", nlayers(sim_features)),
                         which(r3$met), "red"))

## End(Not run)
```

---

add_max_phylo_end_objective
Add maximum phylogenetic endemism objective

Description

Set the objective of a conservation planning problem() to maximize the phylogenetic endemism of the features represented in the solution subject to a budget. This objective is similar to add_max_phylo_end_objective() except that emphasis is placed on representing species with geographically restricted evolutionary histories, instead representing as much evolutionary history as possible. This function was inspired by Faith (1992), Rodrigues *et al.* (2002), and Rosauer *et al.* (2009).

Usage

```
add_max_phylo_end_objective(x, budget, tree)
```

Arguments

x	problem() (i.e. ConservationProblem) object.
budget	numeric value specifying the maximum expenditure of the prioritization. For problems with multiple zones, the argument to budget can be a single numeric value to specify a budget for the entire solution or a numeric vector to specify a budget for each each management zone.
tree	phylo() object specifying a phylogenetic tree for the conservation features.

Details

A problem objective is used to specify the overall goal of the conservation planning problem. Please note that all conservation planning problems formulated in the **prioritizr** package require the addition of objectives—failing to do so will return an error message when attempting to solve problem.

The maximum phylogenetic endemism objective finds the set of planning units that meets representation targets for a phylogenetic tree while staying within a fixed budget. If multiple solutions can meet all targets while staying within budget, the cheapest solution is chosen. Note that this objective is similar to the maximum features objective (`add_max_features_objective()`) in that it allows for both a budget and targets to be set for each feature. However, unlike the maximum feature objective, the aim of this objective is to maximize the total phylogenetic endemism of the targets met in the solution, so if multiple targets are provided for a single feature, the problem will only need to meet a single target for that feature for the phylogenetic benefit for that feature to be counted when calculating the phylogenetic endemism of the solution. In other words, for multi-zone problems, this objective does not aim to maximize the phylogenetic endemism in each zone, but rather this objective aims to maximize the phylogenetic endemism of targets that can be met through allocating planning units to any of the different zones in a problem. This can be useful for problems where targets pertain to the total amount held for each feature across multiple zones. For example, each feature might have a non-zero amount of suitable habitat in each planning unit when the planning units are assigned to a (i) not restored, (ii) partially restored, or (iii) completely restored management zone. Here each target corresponds to a single feature and can be met through the total amount of habitat in planning units present to the three zones.

The maximum phylogenetic endemism objective for the reserve design problem can be expressed mathematically for a set of planning units (I indexed by i) and a set of features (J indexed by j) as:

$$Maximize \sum_{i=1}^{I} -s c_i x_i + \sum_{j=1}^{J} m_b l_b \frac{1}{a_b} subject to \sum_{i=1}^{I} x_i r_{ij} \geq y_j t_j \forall j \in J m_b \leq y_j \forall j \in T(b) \sum_{i=1}^{I} x_i c_i \leq B$$

Here, x_i is the decisions variable (e.g. specifying whether planning unit i has been selected (1) or not (0)), r_{ij} is the amount of feature j in planning unit i, t_j is the representation target for feature j, y_j indicates if the solution has meet the target t_j for feature j. Additionally, T represents a phylogenetic tree containing features j and has the branches b associated within lengths l_b. Each branch $b \in B$ is associated with a total amount a_b indicating the total geographic extent or amount of habitat. The a_b variable for a given branch is calculated by summing the r_{ij} data for all features $j \in J$ that are associated with the branch. The binary variable m_b denotes if at least one feature associated with the branch b has met its representation as indicated by y_j. For brevity, we denote the features j associated with branch b using $T(b)$. Finally, B is the budget allocated for the solution, c_i is the cost of planning unit i, and s is a scaling factor used to shrink the costs so that the problem will return a cheapest solution when there are multiple solutions that represent the same amount of all features within the budget.

Value

Object (i.e. `ConservationProblem`) with the objective added to it.

References

Faith DP (1992) Conservation evaluation and phylogenetic diversity. *Biological Conservation*, 61: 1–10.

Rodrigues ASL and Gaston KJ (2002) Maximising phylogenetic diversity in the selection of networks of conservation areas. *Biological Conservation*, 105: 103–111.

Rosauer D, Laffan SW, Crisp, MD, Donnellan SC and Cook LG (2009) Phylogenetic endemism: a new approach for identifying geographical concentrations of evolutionary history. *Molecular Ecology*, 18: 4061–4072.

See Also

objectives, `branch_matrix()`.

Examples

```
# load ape package
require(ape)

# load data
data(sim_pu_raster, sim_features, sim_phylogeny, sim_pu_zones_stack,
     sim_features_zones)

# plot the simulated phylogeny
## Not run:
par(mfrow = c(1, 1))
plot(sim_phylogeny, main = "phylogeny")

## End(Not run)
# create problem with a maximum phylogenetic endemism objective,
# where each feature needs 10% of its distribution to be secured for
# it to be adequately conserved and a total budget of 1900
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_max_phylo_end_objective(1900, sim_phylogeny) %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s1 <- solve(p1)

# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)

# find out which features have their targets met
r1 <- eval_target_coverage_summary(p1, s1)
print(r1, width = Inf)

# plot the phylogeny and color the adequately represented features in red
plot(sim_phylogeny, main = "adequately represented features",
     tip.color = replace(
       rep("black", nlayers(sim_features)),
       sim_phylogeny$tip.label %in% r1$feature[r1$met],
       "red"))

## End(Not run)
```

```
# rename the features in the example phylogeny for use with the
# multi-zone data
sim_phylogeny$tip.label <- feature_names(sim_features_zones)

# create targets for a multi-zone problem. Here, each feature needs a total
# of 10 units of habitat to be conserved among the three zones to be
# considered adequately conserved
targets <- tibble::tibble(
  feature = feature_names(sim_features_zones),
  zone = list(zone_names(sim_features_zones))[rep(1,
          number_of_features(sim_features_zones))],
  type = rep("absolute", number_of_features(sim_features_zones)),
  target = rep(10, number_of_features(sim_features_zones)))

# create a multi-zone problem with a maximum phylogenetic endemism
# objective, where the total expenditure in all zones is 5000.
p2 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_max_phylo_end_objective(5000, sim_phylogeny) %>%
      add_manual_targets(targets) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s2 <- solve(p2)

# plot solution
plot(category_layer(s2), main = "solution", axes = FALSE, box = FALSE)

# find out which features have their targets met
r2 <- eval_target_coverage_summary(p2, s2)
print(r2, width = Inf)

# plot the phylogeny and color the adequately represented features in red
plot(sim_phylogeny, main = "adequately represented features",
     tip.color = replace(rep("black", nlayers(sim_features)),
                         which(r2$met), "red"))

## End(Not run)
# create a multi-zone problem with a maximum phylogenetic endemism
# objective, where each zone has a separate budget.
p3 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_max_phylo_end_objective(c(2500, 500, 2000), sim_phylogeny) %>%
      add_manual_targets(targets) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s3 <- solve(p3)

# plot solution
plot(category_layer(s3), main = "solution", axes = FALSE, box = FALSE)

# find out which features have their targets met
```

```
r3 <- eval_target_coverage_summary(p3, s3)
print(r3, width = Inf)

# plot the phylogeny and color the adequately represented features in red
plot(sim_phylogeny, main = "adequately represented features",
     tip.color = replace(rep("black", nlayers(sim_features)),
                         which(r3$met), "red"))

## End(Not run)
```

---

add_max_utility_objective
Add maximum utility objective

Description

Set the objective of a conservation planning `problem()` to secure as much of the features as possible without exceeding a budget. This type of objective does not use targets, and feature weights should be used instead to increase the representation of different features in solutions. Note that this objective does not aim to maximize as much of each feature as possible and so often results in solutions that are heavily biased towards specific features.

Usage

```
add_max_utility_objective(x, budget)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object.
budget	`numeric` value specifying the maximum expenditure of the prioritization. For problems with multiple zones, the argument to `budget` can be a single `numeric` value to specify a budget for the entire solution or a `numeric` `vector` to specify a budget for each each management zone.

Details

A problem objective is used to specify the overall goal of the conservation planning problem. Please note that all conservation planning problems formulated in the **prioritizr** package require the addition of objectives—failing to do so will return an error message when attempting to solve problem.

The maximum utility objective seeks to find the set of planning units that maximizes the overall level of representation across a suite of conservation features, while keeping cost within a fixed budget. Additionally, weights can be used to favor the representation of certain features over other features (see `add_feature_weights()`). This objective can be expressed mathematically for a set of planning units (I indexed by i) and a set of features (J indexed by j) as:

$$Maximize \sum_{i=1}^{I} -sc_i x_i + \sum_{j=1}^{J} a_j w_j subjecttoa_j = \sum_{i=1}^{I} x_i r_{ij} \forall j \in J \sum_{i=1}^{I} x_i c_i \leq B$$

Here, x_i is the decisions variable (e.g. specifying whether planning unit i has been selected (1) or not (0)), r_{ij} is the amount of feature j in planning unit i, A_j is the amount of feature j represented in in the solution, and w_j is the weight for feature j (defaults to 1 for all features; see `add_feature_weights()` to specify weights). Additionally, B is the budget allocated for the solution, c_i is the cost of planning unit i, and s is a scaling factor used to shrink the costs so that the problem will return a cheapest solution when there are multiple solutions that represent the same amount of all features within the budget.

Value

Object (i.e. `ConservationProblem`) with the objective added to it.

Notes

In early versions (< 3.0.0.0), this function was named as the `add_max_cover_objective` function. It was renamed to avoid confusion with existing terminology.

See Also

`add_feature_weights()`, objectives.

Examples

```
# load data
data(sim_pu_raster, sim_pu_zones_stack, sim_features, sim_features_zones)

# create problem with maximum utility objective
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_max_utility_objective(5000) %>%
      add_binary_decisions() %>%
      add_default_solver(gap = 0, verbose = FALSE)
## Not run:
# solve problem
s1 <- solve(p1)

# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)

# create multi-zone problem with maximum utility objective that
# has a single budget for all zones
p2 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_max_utility_objective(5000) %>%
      add_binary_decisions() %>%
      add_default_solver(gap = 0, verbose = FALSE)
## Not run:
# solve problem
s2 <- solve(p2)

# plot solution
plot(category_layer(s2), main = "solution", axes = FALSE, box = FALSE)
```

```
## End(Not run)

# create multi-zone problem with maximum utility objective that
# has separate budgets for each zone
p3 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_max_utility_objective(c(1000, 2000, 3000)) %>%
      add_binary_decisions() %>%
      add_default_solver(gap = 0, verbose = FALSE)
## Not run:
# solve problem
s3 <- solve(p3)

# plot solution
plot(category_layer(s3), main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

add_min_largest_shortfall_objective
Add minimum largest shortfall objective

Description

Set the objective of a conservation planning `problem()` to minimize the largest target shortfall while ensuring that the cost of the solution does not exceed a budget. Note that if the target shortfall for a single feature cannot be decreased beyond a certain point (e.g. because all remaining planning units occupied by that feature are too costly or are locked out), then solutions may only use a small proportion of the specified budget.

Usage

```
add_min_largest_shortfall_objective(x, budget)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object.
budget	`numeric` value specifying the maximum expenditure of the prioritization. For problems with multiple zones, the argument to `budget` can be a single `numeric` value to specify a budget for the entire solution or a `numeric` `vector` to specify a budget for each each management zone.

Details

A problem objective is used to specify the overall goal of the conservation planning problem. Please note that all conservation planning problems formulated in the **prioritizr** package require the addition of objectives—failing to do so will return an error message when attempting to solve problem.

The minimum largest shortfall objective aims to find the set of planning units that minimize the largest shortfall for any of the representation targets—that is, the fraction of each target that remains unmet—for as many features as possible while staying within a fixed budget. This objective is different from the minimum shortfall objective (`add_min_shortfall_objective()`) because this objective minimizes the largest (maximum) target shortfall, whereas the minimum shortfall objective minimizes the total (weighted sum) of the target shortfalls. Note that this objective function is not compatible with feature weights (`add_feature_weights()`).

The minimum largest shortfall objective for the reserve design problem can be expressed mathematically for a set of planning units (I indexed by i) and a set of features (J indexed by j) as:

$$Minimize l subject to \sum_{i=1}^{I} x_i r_{ij} + y_j \geq t_j \forall j \in J l \geq \frac{y_j}{t_j} \forall j \in J \sum_{i=1}^{I} x_i c_i \leq B$$

Here, x_i is the decisions variable (e.g. specifying whether planning unit i has been selected (1) or not (0)), r_{ij} is the amount of feature j in planning unit i, and t_j is the representation target for feature j. Additionally, y_j denotes the target shortfall for the target t_j for feature j, and l denotes the largest target shortfall. Furthermore, B is the budget allocated for the solution, c_i is the cost of planning unit i. Note that y_j and s are continuous variables bounded between zero and infinity.

Value

Object (i.e. `ConservationProblem`) with the objective added to it.

See Also

objectives.

Examples

```
# load data
data(sim_pu_raster, sim_pu_zones_stack, sim_features, sim_features_zones)

# create problem with minimum largest shortfall objective
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_largest_shortfall_objective(1800) %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s1 <- solve(p1)

# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)

# create multi-zone problem with minimum largest shortfall objective,
# with 10% representation targets for each feature, and set
# a budget such that the total maximum expenditure in all zones
```

```
# cannot exceed 3000
p2 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_largest_shortfall_objective(3000) %>%
      add_relative_targets(matrix(0.1, ncol = 3, nrow = 5)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s2 <- solve(p2)

# plot solution
plot(category_layer(s2), main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
# create multi-zone problem with minimum largest shortfall objective,
# with 10% representation targets for each feature, and set
# separate budgets for each management zone
p3 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_largest_shortfall_objective(c(3000, 3000, 3000)) %>%
      add_relative_targets(matrix(0.1, ncol = 3, nrow = 5)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s3 <- solve(p3)

# plot solution
plot(category_layer(s3), main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

`add_min_set_objective` *Add minimum set objective*

Description

Set the objective of a conservation planning `problem()` to minimize the cost of the solution whilst ensuring that all targets are met. This objective is similar to that used in *Marxan* and is detailed in Rodrigues *et al.* (2000).

Usage

```
add_min_set_objective(x)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object.

Details

A problem objective is used to specify the overall goal of the conservation planning problem. Please note that all conservation planning problems formulated in the **prioritizr** package require the addition of objectives—failing to do so will return an error message when attempting to solve problem.

In the context of systematic reserve design, the minimum set objective seeks to find the set of planning units that minimizes the overall cost of a reserve network, while meeting a set of representation targets for the conservation features. This objective is equivalent to a simplified *Marxan* reserve design problem with the Boundary Length Modifier (BLM) set to zero.

The minimum set objective for the reserve design problem can be expressed mathematically for a set of planning units (I indexed by i) and a set of features (J indexed by j) as:

$$Minimize \sum_{i=1}^{I} x_i c_i subjectto \sum_{i=1}^{I} x_i r_{ij} \geq T_j \forall j \in J$$

Here, x_i is the decisions variable (e.g. specifying whether planning unit i has been selected (1) or not (0)), c_i is the cost of planning unit i, r_{ij} is the amount of feature j in planning unit i, and T_j is the target for feature j. The first term is the objective function and the second is the set of constraints. In words this says find the set of planning units that meets all the representation targets while minimizing the overall cost.

Value

Object (i.e. `ConservationProblem`) with the objective added to it.

References

Rodrigues AS, Cerdeira OJ, and Gaston KJ (2000) Flexibility, efficiency, and accountability: adapting reserve selection algorithms to more complex conservation problems. *Ecography*, 23: 565–574.

See Also

objectives, targets.

Examples

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

# load data
data(sim_pu_raster, sim_features, sim_pu_zones_stack, sim_features_zones)

# create minimal problem with minimum set objective
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
```

```
s1 <- solve(p1)

# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)

# create multi-zone problem with minimum set objective
targets_matrix <- matrix(rpois(15, 1), nrow = 5, ncol = 3)

p2 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_absolute_targets(targets_matrix) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s2 <- solve(p2)

# plot solution
plot(category_layer(s2), main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

---

add_min_shortfall_objective

Add minimum shortfall objective

Description

Set the objective of a conservation planning `problem()` to minimize the overall shortfall for as many targets as possible while ensuring that the cost of the solution does not exceed a budget.

Usage

```
add_min_shortfall_objective(x, budget)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object.
budget	`numeric` value specifying the maximum expenditure of the prioritization. For problems with multiple zones, the argument to `budget` can be a single `numeric` value to specify a budget for the entire solution or a `numeric` `vector` to specify a budget for each each management zone.

Details

A problem objective is used to specify the overall goal of the conservation planning problem. Please note that all conservation planning problems formulated in the **prioritizr** package require the addition of objectives—failing to do so will return an error message when attempting to solve problem.

The minimum shortfall objective aims to find the set of planning units that minimize the overall (weighted sum) shortfall for the representation targets—that is, the fraction of each target that remains unmet—for as many features as possible while staying within a fixed budget (inspired by Table 1, equation IV, Arponen *et al.* 2005). Additionally, weights can be used to favor the representation of certain features over other features (see `add_feature_weights()`.

The minimum shortfall objective for the reserve design problem can be expressed mathematically for a set of planning units (I indexed by i) and a set of features (J indexed by j) as:

$$Minimize \sum_{j=1}^{J} w_j \frac{y_j}{t_j} subject to \sum_{i=1}^{I} x_i r_{ij} + y_j \geq t_j \forall j \in J \sum_{i=1}^{I} x_i c_i \leq B$$

Here, x_i is the decisions variable (e.g. specifying whether planning unit i has been selected (1) or not (0)), r_{ij} is the amount of feature j in planning unit i, t_j is the representation target for feature j, y_j denotes the representation shortfall for the target t_j for feature j, and w_j is the weight for feature j (defaults to 1 for all features; see `add_feature_weights()` to specify weights). Additionally, B is the budget allocated for the solution, c_i is the cost of planning unit i. Note that y_j is a continuous variable bounded between zero and infinity, and denotes the shortfall for target j.

Value

Object (i.e. `ConservationProblem`) with the objective added to it.

References

Arponen A, Heikkinen RK, Thomas CD, and Moilanen A (2005) The value of biodiversity in reserve selection: representation, species weighting, and benefit functions. *Conservation Biology*, 19: 2009–2014.

See Also

`add_feature_weights()`, objectives.

Examples

```
# load data
data(sim_pu_raster, sim_pu_zones_stack, sim_features, sim_features_zones)

# create problem with minimum shortfall objective
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_shortfall_objective(1800) %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
```

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

# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)

# create multi-zone problem with minimum shortfall objective,
# with 10% representation targets for each feature, and set
# a budget such that the total maximum expenditure in all zones
# cannot exceed 3000
p2 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_shortfall_objective(3000) %>%
      add_relative_targets(matrix(0.1, ncol = 3, nrow = 5)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s2 <- solve(p2)

# plot solution
plot(category_layer(s2), main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
# create multi-zone problem with minimum shortfall objective,
# with 10% representation targets for each feature, and set
# separate budgets for each management zone
p3 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_shortfall_objective(c(3000, 3000, 3000)) %>%
      add_relative_targets(matrix(0.1, ncol = 3, nrow = 5)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s3 <- solve(p3)

# plot solution
plot(category_layer(s3), main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

`add_neighbor_constraints` *Add neighbor constraints*

Description

Add constraints to a conservation planning `problem()` to ensure that all selected planning units in the solution have at least a certain number of neighbors that are also selected in the solution.

Usage

```
## S4 method for signature 'ConservationProblem,ANY,ANY,ANY'
add_neighbor_constraints(x, k, zones, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,data.frame'
add_neighbor_constraints(x, k, zones, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,matrix'
add_neighbor_constraints(x, k, zones, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,array'
add_neighbor_constraints(x, k, zones, data)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object.
k	`integer` minimum number of neighbors for selected planning units in the solution. For problems with multiple zones, the argument to `k` must have an element for each zone.
zones	`matrix` or `Matrix` object describing the neighborhood scheme for different zones. Each row and column corresponds to a different zone in the argument to `x`, and cell values must contain binary `numeric` values (i.e. one or zero) that indicate if neighboring planning units (as specified in the argument to `data`) should be considered neighbors if they are allocated to different zones. The cell values along the diagonal of the matrix indicate if planning units that are allocated to the same zone should be considered neighbors or not. The default argument to `zones` is an identity matrix (i.e. a matrix with ones along the matrix diagonal and zeros elsewhere), so that planning units are only considered neighbors if they are both allocated to the same zone.
data	`NULL`, `matrix`, `Matrix`, `data.frame`, or `array` object showing which planning units are neighbors with each other. The argument defaults to `NULL` which means that the neighborhood data is calculated automatically using the `adjacency_matrix()` function. See the Data format section for more information.

Details

This function uses neighborhood data identify solutions that surround planning units with a minimum number of neighbors. It was inspired by the mathematical formulations detailed in Billionnet (2013) and Beyer *et al.* (2016).

Value

Object (i.e. `ConservationProblem`) with the constraints added to it.

Data format

The argument to `data` can be specified using the following formats:

NULL neighborhood data should be calculated automatically using the adjacency_matrix() function. This is the default argument. Note that the neighborhood data must be manually defined using one of the other formats below when the planning unit data in the argument to x is not spatially referenced (e.g. in data.frame or numeric format).

matrix, Matrix where rows and columns represent different planning units and the value of each cell indicates if the two planning units are neighbors or not. Cell values should be binary numeric values (i.e. one or zero). Cells that occur along the matrix diagonal have no effect on the solution at all because each planning unit cannot be a neighbor with itself.

data.frame containing the fields (columns) "id1", "id2", and "boundary". Here, each row denotes the connectivity between two planning units following the *Marxan* format. The field boundary should contain binary numeric values that indicate if the two planning units specified in the fields "id1" and "id2" are neighbors or not. This data can be used to describe symmetric or asymmetric relationships between planning units. By default, input data is assumed to be symmetric unless asymmetric data is also included (e.g. if data is present for planning units 2 and 3, then the same amount of connectivity is expected for planning units 3 and 2, unless connectivity data is also provided for planning units 3 and 2). If the argument to x contains multiple zones, then the columns "zone1" and "zone2" can optionally be provided to manually specify if the neighborhood data pertain to specific zones. The fields "zone1" and "zone2" should contain the character names of the zones. If the columns "zone1" and "zone2" are present, then the argument to zones must be NULL.

array containing four-dimensions where binary numeric values indicate if planning unit should be treated as being neighbors with every other planning unit when they are allocated to every combination of management zone. The first two dimensions (i.e. rows and columns) correspond to the planning units, and second two dimensions correspond to the management zones. For example, if the argument to data had a value of 1 at the index data[1,2,3,4] this would indicate that planning unit 1 and planning unit 2 should be treated as neighbors when they are allocated to zones 3 and 4 respectively.

References

Beyer HL, Dujardin Y, Watts ME, and Possingham HP (2016) Solving conservation planning problems with integer linear programming. *Ecological Modelling*, 228: 14–22.

Billionnet A (2013) Mathematical optimization ideas for biodiversity conservation. *European Journal of Operational Research*, 231: 514–534.

See Also

constraints.

Examples

```
# load data
data(sim_pu_raster, sim_features, sim_pu_zones_stack, sim_features_zones)

# create minimal problem
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_default_solver(verbose = FALSE)
```

```
# create problem with constraints that require 1 neighbor
# and neighbors are defined using a rook-style neighborhood
p2 <- p1 %>% add_neighbor_constraints(1)

# create problem with constraints that require 2 neighbor
# and neighbors are defined using a rook-style neighborhood
p3 <- p1 %>% add_neighbor_constraints(2)

# create problem with constraints that require 3 neighbor
# and neighbors are defined using a queen-style neighborhood
p4 <- p1 %>% add_neighbor_constraints(3,
              data = adjacency_matrix(sim_pu_raster, directions = 8))

## Not run:
# solve problems
s1 <- stack(list(solve(p1), solve(p2), solve(p3), solve(p4)))

# plot solutions
plot(s1, box = FALSE, axes = FALSE,
     main = c("basic solution", "1 neighbor", "2 neighbors", "3 neighbors"))

## End(Not run)
# create minimal problem with multiple zones
p5 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(0.1, ncol = 3, nrow = 5)) %>%
      add_default_solver(verbose = FALSE)

# create problem where selected planning units require at least 2 neighbors
# for each zone and planning units are only considered neighbors if they
# are allocated to the same zone
z6 <- diag(3)
print(z6)
p6 <- p5 %>% add_neighbor_constraints(rep(2, 3), z6)

# create problem where the planning units in zone 1 don't explicitly require
# any neighbors, planning units in zone 2 require at least 1 neighbors, and
# planning units in zone 3 require at least 2 neighbors. As before, planning
# units are still only considered neighbors if they are allocated to the
# same zone
p7 <- p5 %>% add_neighbor_constraints(c(0, 1, 2), z6)

# create problem given the same constraints as outlined above, except
# that when determining which selected planning units are neighbors,
# planning units that are allocated to zone 1 and zone 2 can also treated
# as being neighbors with each other
z8 <- diag(3)
z8[1, 2] <- 1
z8[2, 1] <- 1
print(z8)
p8 <- p5 %>% add_neighbor_constraints(c(0, 1, 2), z8)
## Not run:
```

```
# solve problems
s2 <- list(p5, p6, p7, p8)
s2 <- lapply(s2, solve)
s2 <- lapply(s2, category_layer)
s2 <- stack(s2)
names(s2) <- c("basic problem", "p6", "p7", "p8")

# plot solutions
plot(s2, main = names(s2), box = FALSE, axes = FALSE)

## End(Not run)
```

`add_proportion_decisions` *Add proportion decisions*

Description

Add a proportion decision to a conservation planning `problem()`. This is a relaxed decision where a part of a planning unit can be prioritized as opposed to the entire planning unit. Typically, this decision has the assumed action of buying a fraction of a planning unit to include in decisions will solve much faster than problems that use binary-type decisions

Usage

```
add_proportion_decisions(x)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object.

Details

Conservation planning problems involve making decisions on planning units. These decisions are then associated with actions (e.g. turning a planning unit into a protected area). If no decision is explicitly added to a problem, then the binary decision class will be used by default. Only a single decision should be added to a `ConservationProblem` object. Note that if multiple decisions are added to a problem object, then the last one to be added will be used.

Value

Object (i.e. `ConservationProblem`) with the decisions added to it.

See Also

decisions.

Examples

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

# load data
data(sim_pu_raster, sim_features, sim_pu_zones_stack, sim_features_zones)

# create minimal problem with proportion decisions
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_proportion_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s1 <- solve(p1)

# plot solutions
plot(s1, main = "solution")

## End(Not run)
# build multi-zone conservation problem with proportion decisions
p2 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_proportion_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve the problem
s2 <- solve(p2)

# print solution
print(s2)

# plot solution
# panels show the proportion of each planning unit allocated to each zone
plot(s2, axes = FALSE, box = FALSE)

## End(Not run)
```

`add_relative_targets` *Add relative targets*

Description

Set targets as a proportion (between 0 and 1) of the maximum level of representation of features in the study area. Please note that proportions are scaled according to the features' total abundances in the study area (including any locked out planning units, or planning units with `NA` cost data) using the `feature_abundances()` function.

Usage

```
add_relative_targets(x, targets)

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

## S4 method for signature 'ConservationProblem,matrix'
add_relative_targets(x, targets)

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

Arguments

`x`	`problem()` (i.e. `ConservationProblem`) object.
`targets`	Object that specifies the targets for each feature. See the Targets data format section for more information.

Details

Targets are used to specify the minimum amount or proportion of a feature's distribution that needs to be protected. Most conservation planning problems require targets with the exception of the maximum cover (see `add_max_cover_objective()`) and maximum utility (see `add_max_utility_objective()`) problems. Attempting to solve problems with objectives that require targets without specifying targets will throw an error.

For problems associated with multiple management zones, this function can be used to set targets that each pertain to a single feature and a single zone. To set targets which can be met through allocating different planning units to multiple zones, see the `add_manual_targets()` function. An example of a target that could be met through allocations to multiple zones might be where each management zone is expected to result in a different amount of a feature and the target requires that the total amount of the feature in all zones must exceed a certain threshold. In other words, the target does not require that any single zone secure a specific amount of the feature, but the total amount held in all zones must secure a specific amount. Thus the target could, potentially, be met through allocating all planning units to any specific management zone, or through allocating the planning units to different combinations of management zones.

Value

Object (i.e. `ConservationProblem`) with the targets added to it.

Targets data format

The targets for a problem can be specified using the following formats.

`numeric` vector of target values for each feature. Additionally, for convenience, this type of argument can be a single value to assign the same target to each feature. Note that this type of argument cannot be used to specify targets for problems with multiple zones.

`matrix` containing a target for each feature in each zone. Here, each row corresponds to a different feature in argument to `x`, each column corresponds to a different zone in argument to `x`, and each cell contains the target value for a given feature that the solution needs to secure in a given zone.

`character` containing the names of fields (columns) in the feature data associated with the argument to `x` that contain targets. This type of argument can only be used when the feature data associated with `x` is a `data.frame`. This argument must contain a field (column) name for each zone.

See Also

targets.

Examples

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

# load data
data(sim_pu_raster, sim_features)

# create base problem
p <- problem(sim_pu_raster, sim_features) %>%
     add_min_set_objective() %>%
     add_binary_decisions() %>%
     add_default_solver(verbose = FALSE)

# create problem with 10% targets
p1 <- p %>% add_relative_targets(0.1)

# create problem with varying targets for each feature
targets <- c(0.1, 0.2, 0.3, 0.4, 0.5)
p2 <- p %>% add_relative_targets(targets)
## Not run:
# solve problem
s <- stack(solve(p1), solve(p2))

# plot solution
plot(s, main = c("10% targets", "varying targets"), axes = FALSE,
     box = FALSE)

## End(Not run)
# create a problem with multiple management zones
p3 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create a problem with targets that specify an equal amount of each feature
# to be represented in each zone
p4_targets <- matrix(0.1, nrow = 5, ncol = 3,
                     dimnames = list(feature_names(sim_features_zones),
```

```
                                       zone_names(sim_features_zones)))
print(p4_targets)

p4 <- p3 %>% add_relative_targets(p4_targets)

# solve problem
## Not run:
# solve problem
s4 <- solve(p4)

# plot solution (pixel values correspond to zone identifiers)
plot(category_layer(s4), main = c("equal targets"))

## End(Not run)
# create a problem with targets that require a varying amount of each
# feature to be represented in each zone
p5_targets <- matrix(runif(15, 0.01, 0.2), nrow = 5, ncol = 3,
                     dimnames = list(feature_names(sim_features_zones),
                                     zone_names(sim_features_zones)))
print(p5_targets)

p5 <- p3 %>% add_relative_targets(p4_targets)
# solve problem
## Not run:
# solve problem
s5 <- solve(p5)

# plot solution (pixel values correspond to zone identifiers)
plot(category_layer(s5), main = c("varying targets"))

## End(Not run)
```

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

Description

Specify that the *SYMPHONY* software (Ralphs & Güzelsoy 2005) – using the **Rsymphony** package – should be used to solve a conservation planning `problem()`. 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 = 0.1,
  time_limit = .Machine$integer.max,
  first_feasible = FALSE,
  verbose = TRUE
)
```

Arguments

- `x` — `problem()` (i.e. `ConservationProblem`) 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 0.1 (i.e. 10% from optimality).
- `time_limit` — `numeric` time limit (seconds) for generating solutions. The solver will return the current best solution when this time limit is exceeded. The default value is the largest integer value (i.e. `.Machine$integer.max`), effectively meaning that solver will keep running until a solution within the optimality gap is found.
- `first_feasible` — `logical` should the first feasible solution be be returned? If `first_feasible` is set to `TRUE`, the solver will return the first solution it encounters that meets all the constraints, regardless of solution quality. Note that the first feasible solution is not an arbitrary solution, rather it is derived from the relaxed solution, and is therefore often reasonably close to optimality. Defaults to `FALSE`.
- `verbose` — `logical` should information be printed while solving optimization problems? Defaults to `TRUE`.

Details

SYMPHONY is an open-source mixed integer programming solver that is part of the Computational Infrastructure for Operations Research (COIN-OR) project. The **Rsymphony** package provides an interface to COIN-OR and – unlike dependencies for other solvers – is available on *CRAN*. For information on the performance of different solvers, please see Schuster *et al.* (2020) for benchmarks comparing the run time and solution quality of different solvers when applied to different sized datasets.

Value

Object (i.e. `ConservationProblem`) with the solver added to it.

References

Ralphs TK and Güzelsoy M (2005) The SYMPHONY callable library for mixed integer programming. In The Next Wave in Computing, Optimization, and Decision Technologies (pp. 61–76). Springer, Boston, MA.

Schuster R, Hanson JO, Strimas-Mackey M, and Bennett JR (2020). Exact integer linear programming solvers outperform simulated annealing for solving conservation planning problems. *PeerJ*, 8: e9258.

See Also

solvers.

Examples

```
## Not run:
# load data
data(sim_pu_raster, sim_features)

# create problem
p <- problem(sim_pu_raster, sim_features) %>%
     add_min_set_objective() %>%
     add_relative_targets(0.1) %>%
     add_binary_decisions() %>%
     add_rsymphony_solver(time_limit = 10, verbose = FALSE)

# generate solution
s <- solve(p)

# plot solution
plot(s, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

---

add_semicontinuous_decisions

Add semi-continuous decisions

Description

Add a semi-continuous decision to a conservation planning `problem()`. This is a relaxed decision where a part of a planning unit can be prioritized, as opposed to the entire planning unit, which is the default function (see `add_binary_decisions()`). This decision is similar to the `add_proportion_decisions()` function except that it has an upper bound parameter. By default, the decision can range from prioritizing none (0%) to all (100%) of a planning unit. However, an upper bound can be specified to ensure that at most only a fraction (e.g. 80%) of a planning unit can be preserved. This type of decision may be useful when it is not practical to conserve entire planning units.

Usage

```
add_semicontinuous_decisions(x, upper_limit)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object.
upper_limit	`numeric` value specifying the maximum proportion of a planning unit that can be reserved (e.g. set to 0.8 for 80%).

Details

Conservation planning problems involve making decisions on planning units. These decisions are then associated with actions (e.g. turning a planning unit into a protected area). If no decision is explicitly added to a problem, then the binary decision class will be used by default. Only a single decision should be added to a `ConservationProblem` object. Note that if multiple decisions are added to a problem object, then the last one to be added will be used.

Value

Object (i.e. `ConservationProblem`) with the decisions added to it.

See Also

decisions.

Examples

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

# load data
data(sim_pu_raster, sim_features, sim_pu_zones_stack, sim_features_zones)

# create minimal problem with semi-continuous decisions
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_semicontinuous_decisions(0.5) %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s1 <- solve(p1)

# plot solutions
plot(s1, main = "solution")

## End(Not run)
# build multi-zone conservation problem with semi-continuous decisions
p2 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_semicontinuous_decisions(0.5) %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve the problem
s2 <- solve(p2)

# print solution
print(s2)
```

```
# plot solution
# panels show the proportion of each planning unit allocated to each zone
plot(s2, axes = FALSE, box = FALSE)

## End(Not run)
```

---

add_shuffle_portfolio *Add a shuffle portfolio*

Description

Generate a portfolio of solutions for a conservation planning `problem()` by randomly reordering the data prior to solving the problem. This is recommended as a replacement for `add_top_portfolio()` when the *Gurobi* software is not available.

Usage

```
add_shuffle_portfolio(
  x,
  number_solutions = 10L,
  threads = 1L,
  remove_duplicates = TRUE
)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object.
number_solutions	`integer` number of attempts to generate different solutions. Defaults to 10.
threads	`integer` number of threads to use for the generating the solution portfolio. Defaults to 1.
remove_duplicates	`logical` should duplicate solutions be removed? Defaults to `TRUE`.

Details

This strategy for generating a portfolio of solutions often results in different solutions, depending on optimality gap, but may return duplicate solutions. In general, this strategy is most effective when problems are quick to solve and multiple threads are available for solving each problem separately.

Value

Object (i.e. `ConservationProblem`) with the portfolio added to it.

See Also

portfolios.

Examples

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

# load data
data(sim_pu_raster, sim_features, sim_pu_zones_stack, sim_features_zones)

# create minimal problem with shuffle portfolio
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_shuffle_portfolio(10, remove_duplicates = FALSE) %>%
      add_default_solver(gap = 0.2, verbose = FALSE)

## Not run:
# solve problem and generate 10 solutions within 20% of optimality
s1 <- solve(p1)

# plot solutions in portfolio
plot(stack(s1), axes = FALSE, box = FALSE)

## End(Not run)
# build multi-zone conservation problem with shuffle portfolio
p2 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_binary_decisions() %>%
      add_shuffle_portfolio(10, remove_duplicates = FALSE) %>%
      add_default_solver(gap = 0.2, verbose = FALSE)

## Not run:
# solve the problem
s2 <- solve(p2)

# print solution
str(s2, max.level = 1)

# plot solutions in portfolio
plot(stack(lapply(s2, category_layer)), main = "solution", axes = FALSE,
     box = FALSE)

## End(Not run)
```

---

add_top_portfolio   *Add a top portfolio*

Description

Generate a portfolio of solutions for a conservation planning problem() by finding a pre-specified number of solutions that are closest to optimality (i.e the top solutions).

Usage

```
add_top_portfolio(x, number_solutions)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object.
number_solutions	`integer` number of solutions required.

Details

This strategy for generating a portfolio requires problems to be solved using the *Gurobi* software suite (i.e. using `add_gurobi_solver()`. Specifically, version 9.0.0 (or greater) of the **gurobi** package must be installed. Note that the number of solutions returned may be less than the argument to `number_solutions`, if the total number of feasible solutions is less than the number of solutions requested.

Value

Object (i.e. `ConservationProblem`) with the portfolio added to it.

See Also

portfolios.

Examples

```
## Not run:
# set seed for reproducibility
set.seed(600)

# load data
data(sim_pu_raster, sim_features)

# create minimal problem with a portfolio for the top 5 solutions
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.05) %>%
      add_top_portfolio(number_solutions = 5) %>%
      add_default_solver(gap = 0, verbose = FALSE)

# solve problem and generate portfolio
s1 <- solve(p1)

# print number of solutions found
print(length(s1))

# plot solutions
plot(stack(s1), axes = FALSE, box = FALSE)

# create multi-zone problem with a portfolio for the top 5 solutions
```

```
p2 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_top_portfolio(number_solutions = 5) %>%
      add_default_solver(gap = 0, verbose = FALSE)

# solve problem and generate portfolio
s2 <- solve(p2)

# print number of solutions found
print(length(s2))

# plot solutions in portfolio
plot(stack(lapply(s2, category_layer)), main = "solution", axes = FALSE,
     box = FALSE)

## End(Not run)
```

---

adjacency_matrix *Adjacency matrix*

Description

Create a matrix showing which planning units are spatially adjacent to each other. Note that this also include planning units that overlap with each other too.

Usage

```
adjacency_matrix(x, ...)

## S3 method for class 'Raster'
adjacency_matrix(x, directions = 4L, ...)

## S3 method for class 'SpatialPolygons'
adjacency_matrix(x, ...)

## S3 method for class 'SpatialLines'
adjacency_matrix(x, ...)

## S3 method for class 'SpatialPoints'
adjacency_matrix(x, ...)

## S3 method for class 'sf'
adjacency_matrix(x, ...)

## Default S3 method:
adjacency_matrix(x, ...)
```

Arguments

x	Raster, SpatialPolygons, SpatialLines, or sf::sf() object representing planning units.
...	not used.
directions	integer If x is a Raster object, the number of directions in which cells should be considered adjacent: 4 (rook's case), 8 (queen's case), 16 (knight and one-cell queen moves), or "bishop" to for cells with one-cell diagonal moves.

Details

Spatial processing is completed using sf::st_intersects() for Spatial and sf::sf() objects, and raster::adjacent() for Raster objects.

Value

dsCMatrix sparse symmetric matrix. Each row and column represents a planning unit. Cells values indicate if different planning units are adjacent to each other or not (using ones and zeros). To reduce computational burden, cells among the matrix diagonal are set to zero. Furthermore, if the argument to x is a Raster object, then cells with NA values are set to zero too.

Notes

In earlier versions (< 5.0.0), this function was named as the connected_matrix function. It has been renamed to be consistent with other spatial association matrix functions.

Examples

```
# load data
data(sim_pu_raster, sim_pu_sf, sim_pu_lines)

# create adjacency matrix using raster data
## crop raster to 9 cells
r <- crop(sim_pu_raster, c(0, 0.3, 0, 0.3))

## make adjacency matrix
am_raster <- adjacency_matrix(r)

# create adjacency matrix using polygons (sf) data
## subset 9 polygons
ply <- sim_pu_sf[c(1:2, 10:12, 20:22), ]

## make adjacency matrix
am_ply <- adjacency_matrix(ply)

# create adjacency matrix using lines (Spatial) data
## subset 9 lines
lns <- sim_pu_lines[c(1:2, 10:12, 20:22), ]

## make adjacency matrix
am_lns <- adjacency_matrix(lns)
```

```
# plot data and the adjacency matrices
## Not run:
par(mfrow = c(4,2))

## plot raster and adjacency matrix
plot(r, main = "raster", axes = FALSE, box = FALSE)
plot(raster(as.matrix(am_raster)), main = "adjacency matrix", axes = FALSE,
     box = FALSE)

## plot polygons (sf) and adjacency matrix
plot(r, main = "polygons (sf)", axes = FALSE, box = FALSE)
plot(raster(as.matrix(am_ply)), main = "adjacency matrix", axes = FALSE,
    box = FALSE)

## plot lines (Spatial) and adjacency matrix
plot(r, main = "lines (Spatial)", axes = FALSE, box = FALSE)
plot(raster(as.matrix(am_lns)), main = "adjacency matrix", axes = FALSE,
     box = FALSE)

## End(Not run)
```

`ArrayParameter-class` *Array parameter prototype*

Description

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

Fields

$id `character` identifier for parameter.

$name `character` name of parameter.

$value `numeric` `vector` of values.

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

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

$length `integer` number of values.

$class `character` class of values.

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

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

Usage

```
x$print()

x$show()

x$repr()

x$validate(tbl)

x$get()

x$set(tbl)

x$reset()
```

Arguments

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

Details

print print the object.

show show the object.

repr `character` representation of object.

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

get return a `base::data.frame()` containing the parameter values.

set update the parameter values using a `base::data.frame()`.

reset update the parameter values to be the default values.

See Also

`ScalarParameter`, `Parameter`.

`array_parameters` *Array parameters*

Description

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

Usage

```
proportion_parameter_array(name, value, label)

binary_parameter_array(name, value, label)

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

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

Arguments

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

Details

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

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

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

integer_parameter_array a parameter that consists of multiple `integer` values.

numeric_parameter_array a parameter that consists of multiple `numeric` values.

Value

`ArrayParameter` object.

Examples

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

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

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

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

# numeric parameter array with lower bounds
p5 <- numeric_parameter_array('b_dbl_array', c(0.1, 4, -5), letters[1:3],
```

```
                              lower_limit=c(0, 1, 2))
print(p5) # print it
p5$get() # get value
p5$id# get id
invalid <- data.frame(value = c(-1, 5, 5), row.names=letters[1:3]) # invalid
p5$validate(invalid) # check invalid input is invalid
valid <- data.frame(value = c(0, 1, 2), row.names=letters[1:3]) # valid
p5$validate(valid) # check valid input is valid
p5$set(valid) # change value to valid input
print(p5)
```

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

Description

Coerce an object.

Usage

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

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

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

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

Arguments

`x`	Object.
`...`	unused arguments.

Value

An object.

as.list.OptimizationProblem

Convert `OptimizationProblem` *to list*

Description

Convert `OptimizationProblem` to list

Usage

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

Arguments

x	`OptimizationProblem` object.
...	not used.

Value

`list()` object.

binary_stack

Binary stack

Description

Convert a `RasterLayer` object containing categorical identifiers into a `RasterStack` object where each layer corresponds to a different identifier and values indicate the presence/absence of that category in the input object.

Usage

```
binary_stack(x)
```

Arguments

x	`Raster` object containing a single layer.

Details

This function is provided to help manage data that encompass multiple management zones. For instance, this function may be helpful for preparing raster data for `add_locked_in_constraints()` and `add_locked_out_constraints()` since they require binary `RasterStack` objects as input arguments.

Value

RasterStack object.

See Also

category_layer().

Examples

```
# create raster with categorical identifers
x <- raster(matrix(c(1, 2, 3, 1, NA, 1), nrow = 3))

# convert to binary stack
y <- binary_stack(x)

# plot categorical raster and binary stack representation
## Not run:
plot(stack(x, y), main = c("x", "y[[1]]", "y[[2]]", "y[[3]]"), nr = 1)

## End(Not run)
```

---

boundary_matrix *Boundary matrix*

Description

Generate a matrix describing the amount of shared boundary length between different planning units, and the amount of exposed edge length each planning unit exhibits.

Usage

```
boundary_matrix(x, str_tree)

## S3 method for class 'Raster'
boundary_matrix(x, str_tree = FALSE)

## S3 method for class 'SpatialPolygons'
boundary_matrix(x, str_tree = FALSE)

## S3 method for class 'SpatialLines'
boundary_matrix(x, str_tree = FALSE)

## S3 method for class 'SpatialPoints'
boundary_matrix(x, str_tree = FALSE)

## S3 method for class 'sf'
boundary_matrix(x, str_tree = FALSE)
```

```
## Default S3 method:
boundary_matrix(x, str_tree = FALSE)
```

Arguments

x Raster, SpatialLines, SpatialPolygons, sf::sf() object representing planning units. If x is a Raster object then it must have only one layer.

str_tree logical should a GEOS STRtree structure be used to to pre-process data? If TRUE, then the experimental rgeos::gUnarySTRtreeQuery() function will be used to pre-compute which planning units are adjacent to each other and potentially reduce the processing time required to generate the boundary matrices. This argument is only used when the planning unit data are vector-based polygons (i.e. sp::SpatialPolygonsDataFrame() objects). **Note that using TRUE may crash Mac OSX systems.** The default argument is FALSE.

Details

This function returns a dsCMatrix symmetric sparse matrix. Cells on the off-diagonal indicate the length of the shared boundary between two different planning units. Cells on the diagonal indicate length of a given planning unit's edges that have no neighbors (e.g. for edges of planning units found along the coastline). **This function assumes the data are in a coordinate system where Euclidean distances accurately describe the proximity between two points on the earth**. Thus spatial data in a longitude/latitude coordinate system (i.e. WGS84) should be reprojected to another coordinate system before using this function. Note that for Raster objects boundaries are missing for cells that have NA values in all cells.

Value

dsCMatrix symmetric sparse matrix object. Each row and column represents a planning unit. Cells values indicate the shared boundary length between different pairs of planning units.

Examples

```
# load data
data(sim_pu_raster, sim_pu_polygons)

# subset data to reduce processing time
r <- crop(sim_pu_raster, c(0, 0.3, 0, 0.3))
ply <- sim_pu_polygons[c(1:2, 10:12, 20:22), ]
ply2 <- st_as_sf(ply)

# create boundary matrix using raster data
bm_raster <- boundary_matrix(r)

# create boundary matrix using polygon (Spatial) data
bm_ply1 <- boundary_matrix(ply)

# create boundary matrix using polygon (sf) data
bm_ply2 <- boundary_matrix(ply2)

# create boundary matrix with polygon (Spatial) data and GEOS STR query trees
```

```
# to speed up processing
bm_ply3 <- boundary_matrix(ply, TRUE)

# plot raster and boundary matrix
## Not run:
par(mfrow = c(1, 2))
plot(r, main = "raster", axes = FALSE, box = FALSE)
plot(raster(as.matrix(bm_raster)), main = "boundary matrix",
     axes = FALSE, box = FALSE)

## End(Not run)
# plot polygons and boundary matrices
## Not run:
par(mfrow = c(1, 3))
plot(r, main = "polygons (Spatial)", axes = FALSE, box = FALSE)
plot(raster(as.matrix(bm_ply1)), main = "boundary matrix", axes = FALSE,
     box = FALSE)
plot(r, main = "polygons (sf)", axes = FALSE, box = FALSE)
plot(raster(as.matrix(bm_ply2)), main = "boundary matrix", axes = FALSE,
     box = FALSE)
plot(raster(as.matrix(bm_ply3)), main = "boundary matrix (Spatial, STR)",
            axes = FALSE, box = FALSE)

## End(Not run)
```

`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 both share that same period of evolutionary history. This function creates a matrix that shows which species are connected with branch. In other words, it creates a matrix that shows which periods of evolutionary history each species have experienced.

Usage

```
branch_matrix(x)

## Default S3 method:
branch_matrix(x)

## S3 method for class 'phylo'
branch_matrix(x)
```

Arguments

x	`ape::phylo()` tree object.

Value

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 denoted with a one.

Examples

```
# load data
data(sim_phylogeny)

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

# plot data
## Not run:
par(mfrow = c(1,2))
plot(sim_phylogeny, main = "phylogeny")
plot(raster(as.matrix(m)), main = "branch matrix", axes = FALSE,
     box = FALSE)

## End(Not run)
```

---

category_layer *Category layer*

Description

Convert a RasterStack object where each layer corresponds to a different identifier and values indicate the presence/absence of that category into a RasterLayer object containing categorical identifiers.

Usage

```
category_layer(x)
```

Arguments

x	Raster object containing a multiple layers. Note that pixels must be 0, 1 or NA values.

Details

This function is provided to help manage data that encompass multiple management zones. For instance, this function may be helpful for interpreting solutions for problems associated with multiple zones that have binary decisions.

Value

RasterLayer object.

See Also

binary_stack().

Examples

```
# create a binary raster stack
x <- stack(raster(matrix(c(1, 0, 0, 1, NA, 0), nrow = 3)),
           raster(matrix(c(0, 1, 0, 0, NA, 0), nrow = 3)),
           raster(matrix(c(0, 0, 1, 0, NA, 1), nrow = 3)))

# convert to binary stack
y <- category_layer(x)

# plot categorical raster and binary stack representation
## Not run:
plot(stack(x, y), main = c("x[[1]]", "x[[2]]", "x[[3]]", "y"), nr = 1)

## End(Not run)
```

---

category_vector   *Category vector*

Description

Convert an object containing binary (integer) fields (columns) into a integer vector indicating the column index where each row is 1.

Usage

```
category_vector(x)

## S3 method for class 'data.frame'
category_vector(x)

## S3 method for class 'sf'
category_vector(x)

## S3 method for class 'Spatial'
category_vector(x)

## S3 method for class 'matrix'
category_vector(x)
```

Arguments

x	matrix, data.frame, Spatial, or sf::sf() object.

Details

This function is conceptually similar to `base::max.col()` except that rows with no values equal to 1 values are assigned a value of zero. Also, note that in the argument to `x`, each row must contain only a single value equal to 1.

Value

`integer` vector

See Also

`base::max.col()`

Examples

```
# create matrix with logical fields
x <- matrix(c(1, 0, 0, NA, 0, 1, 0, NA, 0, 0, 0, NA), ncol = 3)

# print matrix
print(x)

# convert to category vector
y <- category_vector(x)

# print category vector
print(y)
```

Collection-class *Collection prototype*

Description

This prototype represents a collection of `ConservationModifier` objects.

Fields

$... `ConservationModifier` objects stored in the collection.

Usage

`x$print()`

`x$show()`

`x$repr()`

`x$ids()`

`x$length()`

`x$add`

```
x$remove(id)
x$get_parameter(id)
x$set_parameter(id,value)
```

Arguments

id `id` object.

value any object.

Details

print print the object.

show show the object.

repr `character` representation of object.

ids `character` ids for objects inside collection.

length `integer` number of objects inside collection.

find `character` id for object inside collection which contains the input id.

find_parameter `character` id for object inside collection which contains the input `character` object as a parameter.

add add `ConservationModifier` object.

remove remove an item from the collection.

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

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

See Also

`Constraint`, `Penalty`.

---

compile — *Compile a problem*

Description

Compile a conservation planning `problem()` into an (potentially mixed) integer linear programming problem.

Usage

```
compile(x, ...)

## S3 method for class 'ConservationProblem'
compile(x, compressed_formulation = NA, ...)
```

Arguments

x `problem()` (i.e. `ConservationProblem`) object.

... not used.

compressed_formulation
: `logical` should the conservation problem compiled into a compressed version of a planning problem? If `TRUE` then the problem is expressed using the compressed formulation. If `FALSE` then the problem is expressed using the expanded formulation. If `NA`, then the compressed is used unless one of the constraints requires the expanded formulation. This argument defaults to `NA`.

Details

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

Please note that in nearly all cases, the default argument to `formulation` should be used. The only situation where manually setting the argument to `formulation` is desirable is during testing. Manually setting the argument to `formulation` will at best have no effect on the problem. At worst, it may result in an error, a misspecified problem, or unnecessarily long solve times.

Value

`OptimizationProblem` object.

Examples

```
# build minimal conservation problem
p <- problem(sim_pu_raster, sim_features) %>%
  add_min_set_objective() %>%
  add_relative_targets(0.1)

# compile the conservation problem into an optimization problem
o <- compile(p)

# print the optimization problem
print(o)
```

---

connectivity_matrix *Connectivity matrix*

Description

Create a matrix showing the connectivity between planning units. Connectivity is calculated as the average conductance of two planning units multiplied by the amount of shared boundary between the two planning units. Thus planning units that each have higher a conductance and share a greater boundary are associated with greater connectivity.

Usage

```
connectivity_matrix(x, y, ...)

## S4 method for signature 'Spatial,Raster'
connectivity_matrix(x, y, ...)

## S4 method for signature 'Spatial,character'
connectivity_matrix(x, y, ...)

## S4 method for signature 'sf,character'
connectivity_matrix(x, y, ...)

## S4 method for signature 'sf,Raster'
connectivity_matrix(x, y, ...)

## S4 method for signature 'Raster,Raster'
connectivity_matrix(x, y, ...)
```

Arguments

x	Raster, SpatialPolygonsDataFrame, SpatialLinesDataFrame, or sf::sf() object representing planning units. If x is a Raster object then it must contain a single layer.
y	Raster object showing the conductance of different areas across the study area, or a character object denoting a column name in the attribute table of x that contains the conductance values. Note that argument to y can only be a character object if the argument to x is a Spatial or sf::sf() object. Also, note that if the argument to x is a Raster object then argument to y must have the same spatial properties as it (i.e. coordinate system, extent, resolution).
...	additional arguments passed to fast_extract() for extracting and calculating the conductance values for each planning unit. These arguments are only used if argument to x is a link[sp]{Spatial-class} or sf::sf() object and argument to y is a Raster object.

Details

Shared boundary calculations are performed using boundary_matrix().

Value

dsCMatrix symmetric sparse matrix object. Each row and column represents a planning unit. Cells values indicate the connectivity between different pairs of planning units. To reduce computational burden, cells among the matrix diagonal are set to zero. Furthermore, if the argument to x is a Raster object, then cells with NA values are set to zero too.

Examples

```
# load data
```

```
data(sim_pu_raster, sim_pu_sf, sim_features)

# create connectivity matrix using raster planning unit data using
# the raster cost values to represent conductance
## extract 9 planning units
r <- crop(sim_pu_raster, c(0, 0.3, 0, 0.3))

## extract conductance data for the 9 planning units
cd <- crop(sim_features, r)

## make connectivity matrix using the habitat suitability data for the
## second feature to represent the planning unit conductance data
cm_raster <- connectivity_matrix(r, cd[[2]])

## plot data and matrix
## Not run:
par(mfrow = c(1,3))
plot(r, main = "planning units (raster)", axes = FALSE, box = FALSE)
plot(cd[[2]], main = "conductivity", axes = FALSE, box = FALSE)
plot(clamp(raster(as.matrix(cm_raster)), lower = 1e-5, useValues = FALSE),
     main = "connectivity", axes = FALSE, box = FALSE)

## End(Not run)
# create connectivity matrix using polygon planning unit data using
# the habitat suitability data for the second feature to represent
# planning unit conductances
## subset data to 9 polygons
ply <- sim_pu_sf[c(1:2, 10:12, 20:22), ]

## make connectivity matrix
cm_ply <- connectivity_matrix(ply, sim_features[[2]])

## plot data and matrix
## Not run:
par(mfrow = c(1, 2))
plot(st_geometry(ply), main = "planning units (sf)")
plot(clamp(raster(as.matrix(cm_ply)), lower = 1e-5, useValues = FALSE),
     main = "connectivity", axes = FALSE, box = FALSE)

## End(Not run)

# create connectivity matrix using habitat suitability data for each feature,
# this could be useful if prioritisations should spatially clump
# together adjacent planning units that have suitable habitat
# for the same species (e.g. to maintain functional connectivity)

## let's use the raster data for this example, and we can generate the
## connectivity matrix that we would use in the prioritization by
## (1) generating a connectivity matrix for each feature separately, and
## and then (2) then summing the values together
cm_sum <- lapply(as.list(cd), connectivity_matrix, x = r) # make matrices
cm_sum <- Reduce("+", cm_sum) # sum matrices together
```

```
## plot data and matrix
## Not run:
par(mfrow = c(1, 2))
plot(r, main = "planning units (raster)", axes = FALSE, box = FALSE)
plot(clamp(raster(as.matrix(cm_sum)), lower = 1e-5, useValues = FALSE),
     main = "connectivity", axes = FALSE, box = FALSE)

## End(Not run)

## we could take this example one step further, and use weights to indicate
## relative importance of maintaining functional connectivity
## for each feature (i.e. use the weighted sum instead of the sum)

## let's pretend that the first feature is 20 times more important
## than all the other species
weights <- c(20, 1, 1, 1, 1)

## calculate connectivity matrix using weighted sum
cm_wsum <- lapply(as.list(cd), connectivity_matrix, x = r) # make matrices
cm_wsum <- Map("*", cm_wsum, weights) # multiply by weights
cm_wsum <- Reduce("+", cm_wsum) # sum matrices together

## plot data and matrix
## Not run:
par(mfrow = c(1, 2))
plot(r, main = "planning units (raster)", axes = FALSE, box = FALSE)
plot(clamp(raster(as.matrix(cm_wsum)), lower = 1e-5, useValues = FALSE),
     main = "connectivity", axes = FALSE, box = FALSE)

## End(Not run)

## since the statistical distribution of the connectivity values
## for each feature (e.g. the mean and standard deviation of the
## connectivity values) are different, it might make sense -- depending
## on the goal of the conservation planning exercise and the underlying
## data -- to first normalize the conductance values before applying the
## weights and summing the data for feature together

## one approach would be to linearly rescale the values between 0.01 and 1
## note that we wouldn't want to rescale them between 0 and 1 since
## a value of zero means that there is no connectivity at all (and
## and not a relatively small amount of connectivity)
## Not run:
### define helper function
library(scales) # load scales library for rescale
rescale_matrix <- function(x) {x@x <- rescale(x@x, c(0.01, 1)); x}

### calculate functional connectivity matrix using the weighted sum of
### connectivity values that have been normalized by linearly re-scaling
### values
cm_lwsum <- lapply(as.list(cd), connectivity_matrix, x = r) # make matrices
cm_lwsum <- lapply(cm_lwsum, rescale_matrix) # rescale matrices to [0.01, 1]
cm_lwsum <- Map("*", cm_lwsum, weights) # multiply by weights
```

```
cm_lwsum <- Reduce("+", cm_lwsum) # sum matrices together

## End(Not run)

## plot data and matrix
## Not run:
par(mfrow = c(1, 2))
plot(r, main = "planning units (raster)", axes = FALSE, box = FALSE)
plot(clamp(raster(as.matrix(cm_lwsum)), lower = 1e-5, useValues = FALSE),
     main = "connectivity", axes = FALSE, box = FALSE)

## End(Not run)

## another approach for normalizing the data could be using z-scores
## note that after normalizing the data we would need to add a constant
## value so that none of the connectivity values are negative

### define helper functions
zscore <- function(x) {x@x <- (x@x - mean(x@x)) / sd(x@x); x}
min_non_zero_value <- function(x) min(x@x)
add_non_zero_value <- function(x, y) {x@x <- x@x + y; x}

### calculate functional connectivity matrix using the weighted sum of
### connectivity values that have been normalized using z-scores,
### and transformed to account for negative values
cm_zwsum <- lapply(as.list(cd), connectivity_matrix, x = r) # make matrices
cm_zwsum <- lapply(cm_zwsum, zscore) # normalize using z-scores
min_value <- min(sapply(cm_zwsum, min_non_zero_value)) # find min value
min_value <- abs(min_value) + 0.01 # prepare constant for adding to matrices
cm_zwsum <- lapply(cm_zwsum, add_non_zero_value, min_value) # add constant
cm_zwsum <- Map("*", cm_zwsum, weights) # multiply by weights
cm_zwsum <- Reduce("+", cm_zwsum) # sum matrices together

## plot data and matrix
## Not run:
par(mfrow = c(1, 2))
plot(r, main = "planning units (raster)", axes = FALSE, box = FALSE)
plot(clamp(raster(as.matrix(cm_zwsum)), lower = 1e-5, useValues = FALSE),
     main = "connectivity", axes = FALSE, box = FALSE)

## End(Not run)
```

ConservationModifier-class

Conservation problem modifier prototype

Description

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

Fields

$name `character` name of object.

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

$data `list` object with data.

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

Usage

`x$print()`

`x$show()`

`x$repr()`

`x$get_data(name)`

`x$set_data(name,value)`

`x$calculate(cp)`

`x$output()`

`x$apply(op,cp)`

`x$get_parameter(id)`

`x$get_all_parameters()`

`x$set_parameter(id,value)`

Arguments

name `character` name for object

value any object

id `id` or `name` of parameter

cp `ConservationProblem` object

op `OptimizationProblem` object

Details

print print the object.

show show the object.

repr return `character` representation of the object.

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

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

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

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

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

get_parameter retrieve the value of a parameter.

get_all_parameters generate `list` containing all the parameters.

set_parameter change the value of a parameter to new value.

`ConservationProblem-class`
Conservation problem class

Description

This class is used to represent conservation planning problems. A conservation planning problem has spatially explicit planning units. A prioritization involves making a decision on each planning unit (e.g. is the planning unit going to be turned into a protected area?). Each planning unit is associated with a cost that represents the cost incurred by applying the decision to the planning unit. The problem also has a set of representation targets for each feature. Further, it also has constraints used to ensure that the solution meets additional objectives (e.g. certain areas are locked into the solution). Finally, a conservation planning problem—unlike an optimization problem—also requires a method to solve the problem. **This class represents a planning problem, to actually build and then solve a planning problem, use the `problem()` function. Only experts should use this class directly.**

Fields

$data `list` object containing data.

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

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

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

$penalties `Collection` object used to represent additional penalties that the problem is subject to.

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

$portfolio `Portfolio` object used to represent the method for generating a portfolio of solutions.

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

Usage

```
x$print()

x$show()

x$repr()

x$get_data(name)

x$set_data(name,value)

x$number_of_total_units()

x$number_of_planning_units()

x$planning_unit_indices()

x$planning_unit_indices_with_finite_costs()

x$planning_unit_costs()

x$number_of_features()

x$feature_names()

x$feature_abundances_in_planning_units()

x$feature_abundances_in_total_units()

x$feature_targets()

x$number_of_zones()

x$zone_names()

x$add_objective(obj)

x$add_decisions(dec)

x$add_portfolio(pol)

x$add_solver(sol)

x$add_constraint(con)

x$add_targets(targ)

x$get_constraint_parameter(id)

x$set_constraint_parameter(id,value)

x$get_objective_parameter(id)

x$set_objective_parameter(id,value)

x$get_solver_parameter(id)

x$set_solver_parameter(id,value)

x$get_portfolio_parameter(id)

x$set_portfolio_parameter(id,value)

x$get_penalty_parameter(id)

x$set_penalty_parameter(id,value)
```

Arguments

name character name for object.

value an object.

obj Objective object.

dec Decision object.

con Constraint object.

pol Portfolio object.

sol Solver object.

targ Target object.

cost RasterLayer, SpatialPolygonsDataFrame, or SpatialLinesDataFrame object showing spatial representation of the planning units and their cost.

features Zones or data.frame object containing feature data.

id Id object that refers to a specific parameter.

value object that the parameter value should become.

Details

print print the object.

show show the object.

repr return character representation of the object.

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

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

number_of_planning_units integer number of planning units.

planning_unit_indices integer indices of the planning units in the planning unit data.

planning_unit_indices_with_finite_costs list of integer indices of planning units in each zone that have finite cost data.

number_of_total_units integer number of units in the cost data including units that have N cost data.

planning_unit_costs matrix cost of allocating each planning unit to each zone. Each column corresponds to a different zone and each row corresponds to a different planning unit.

number_of_features integer number of features.

feature_names character names of features in problem.

feature_abundances_in_planning_units matrix total abundance of each feature in planning units available in each zone. Each column corresponds to a different zone and each row corresponds to a different feature.

feature_abundances_in_total_units matrix total abundance of each feature in each zone. Each column corresponds to a different zone and each row corresponds to a different feature.

feature_targets tibble::tibble() with feature targets.

number_of_zones integer number of zones.

zone_names character names of zones in problem.

add_objective return a new ConservationProblem with the objective added to it.

add_decisions return a new ConservationProblem object with the decision added to it.

add_portfolio return a new ConservationProblem object with the portfolio method added to it.

add_solver return a new ConservationProblem object with the solver added to it.

add_constraint return a new ConservationProblem object with the constraint added to it.

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

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

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

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

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

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

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

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

set_portfolio_parameter set the value of a parameter (specified by argument id) used in objects' solver to value.

Constraint-class *Constraint prototype*

Description

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

See Also

ConservationModifier.

constraints *Conservation problem constraints*

Description

A constraint can be added to a conservation planning `problem()` to ensure that solutions exhibit a specific characteristic.

Details

Constraints can be used to ensure that solutions exhibit a range of different characteristics. For instance, they can be used to lock in or lock out certain planning units from the solution, such as protected areas or degraded land (respectively). Additionally, similar to the penalties functions, some of the constraint functions can be used to increase connectivity in a solution. The key difference between a penalty and a constraint, however, is that constraints work by invalidating solutions that do not exhibit a specific characteristic, whereas penalty functions work by than penalizing solutions which do not meet a specific characteristic. Thus constraints do not affect the objective function. The following constraints are available.

The following constraints can be added to a conservation planning `problem()`:

`add_locked_in_constraints()` Add constraints to ensure that certain planning units are selected in the solution.

`add_locked_out_constraints()` Add constraints to ensure that certain planning units are not selected in the solution.

`add_neighbor_constraints()` Add constraints to ensure that all selected planning units have at least a certain number of neighbors.

`add_contiguity_constraints()` Add constraints to a ensure that all selected planning units are spatially connected to each other and form a single contiguous unit.

`add_feature_contiguity_constraints()` Add constraints to ensure that each feature is represented in a contiguous unit of dispersible habitat. These constraints are a more advanced version of those implemented in the `add_contiguity_constraints()` function, because they ensure that each feature is represented in a contiguous unit and not that the entire solution should form a contiguous unit.

`add_mandatory_allocation_constraints()` Add constraints to ensure that every planning unit is allocated to a management zone in the solution. **This function can only be used with problems that contain multiple zones.**

See Also

decisions, objectives, penalties, portfolios, `problem()`, solvers, targets.

Examples

```
# load data
data(sim_pu_raster, sim_features, sim_locked_in_raster,
     sim_locked_out_raster)

# create minimal problem with only targets and no additional constraints
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create problem with locked in constraints
p2 <- p1 %>% add_locked_in_constraints(sim_locked_in_raster)

# create problem with locked in constraints
p3 <- p1 %>% add_locked_out_constraints(sim_locked_out_raster)

# create problem with neighbor constraints
p4 <- p1 %>% add_neighbor_constraints(2)

# create problem with contiguity constraints
p5 <- p1 %>% add_contiguity_constraints()

# create problem with feature contiguity constraints
p6 <- p1 %>% add_feature_contiguity_constraints()
## Not run:
# solve problems
s <- stack(lapply(list(p1, p2, p3, p4, p5, p6), solve))

# plot solutions
plot(s, box = FALSE, axes = FALSE, nr = 2,
     main = c("minimal problem", "locked in", "locked out",
              "neighbor", "contiguity", "feature contiguity"))

## End(Not run)
```

`Decision-class` *Decision prototype*

Description

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

See Also

`ConservationModifier`.

`decisions` *Add decision types*

Description

Conservation planning problems involve making decisions on how different planning units will be managed. These decisions might involve turning an entire planning unit into a protected area, turning part of a planning unit into a protected area, or allocating a planning unit to a specific management zone. If no decision is explicitly added to a `problem()`, then binary decisions will be used by default.

Details

Only a single type of decision can be added to a conservation planning `problem()`. Note that if multiple decisions are added to a problem, then the last one added will be used.

The following decisions can be added to a conservation planning `problem()`:

`add_binary_decisions()` Add a binary decision to a conservation planning `problem`. This is the classic decision of either prioritizing or not prioritizing a planning unit. Typically, this decision has the assumed action of buying the planning unit to include in a protected area network. If no decision is added to a problem object then this decision class will be used by default.

`add_proportion_decisions()` Add a proportion decision to a conservation planning `problem`. This is a relaxed decision where a part of a planning unit can be prioritized, as opposed to the default of the entire planning unit. Typically, this decision has the assumed action of buying a fraction of a planning unit to include in a protected area network.

`add_semicontinuous_decisions()` Add a semi-continuous decision to a conservation planning `problem`. This decision is similar to `add_proportion_decision` except that it has an upper bound parameter. By default, the decision can range from prioritizing none (0%) to all (100%) of a planning unit. However, a upper bound can be specified to ensure that at most only a fraction (e.g. 80%) of a planning unit can be preserved. This type of decision may be useful when it is not practical to conserve the entire area encompassed by any single planning unit.

See Also

constraints, objectives, penalties, portfolios, `problem()`, solvers, targets.

Examples

```
# load data
data(sim_pu_raster, sim_features)

# create basic problem and using the default decision types (binary)
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_default_solver(verbose = FALSE)
```

```
# create problem with manually specified binary decisions
p2 <- p1 %>% add_binary_decisions()

# create problem with proportion decisions
p3 <- p1 %>% add_proportion_decisions()

# create problem with semicontinuous decisions
p4 <- p1 %>% add_semicontinuous_decisions(upper_limit = 0.5)

## Not run:
# solve problem
s <- stack(solve(p1), solve(p2), solve(p3), solve(p4))

# plot solutions
plot(s, main = c("default (binary)", "binary", "proportion",
                 "semicontinuous (upper = 0.5)"))

## End(Not run)
```

distribute_load *Distribute load*

Description

Utility function for distributing computations among a pool of workers for parallel processing.

Usage

```
distribute_load(x, n = 1)
```

Arguments

x	integer number of item to process.
n	integer number of threads.

Details

This function returns a list containing an element for each worker. Each element contains a integer vector specifying the indices that the worker should process.

Value

list object.

Examples

```
## Not run:

# imagine that we have 10 jobs that need processing. For simplicity,
# our jobs will involve adding 1 to each element in 1:10.
values <- 1:10

# we could complete this processing using the following vectorized code
result <- 1 + 1:10
print(result)

# however, if our jobs were complex then we would be better off using
# functionals
result <- lapply(1:10, function(x) x + 1)
print(result)

# we could do one better, and use the "plyr" package to handle the
# processing
result <- plyr::llply(1:10, function(x) x + 1)
print(result)

# we could also use the parallel processing options available through "plyr"
# to use more computation resources to complete the jobs (note that since
# these jobs are very quick to process this is actually slower).
cl <- parallel::makeCluster(2, "PSOCK")
doParallel::registerDoParallel(cl)
result <- plyr::llply(1:10, function(x) x + 1, .parallel = TRUE)
cl <- parallel::stopCluster(cl)
print(result)

# however this approach iterates over each element individually, we could
# use the distribute_load function to split the N jobs up into K super
# jobs, and evaluate each super job using vectorized code.
x <- 1:10
cl <- parallel::makeCluster(2, "PSOCK")
parallel::clusterExport(cl, 'x', envir = environment())
doParallel::registerDoParallel(cl)
l <- distribute_load(length(x), n = 2)
result <- plyr::llply(l, function(i) x[i] + 1, .parallel = TRUE)
cl <- parallel::stopCluster(cl)
print(result)

## End(Not run)
```

eval_boundary_summary *Evaluate solution boundary length*

Description

Calculate the exposed boundary length (perimeter) associated with a solution to a conservation planning problem(). This summary statistic is useful for evaluating the spatial fragmentation of planning units selected within a solution.

Usage

```
eval_boundary_summary(x, ...)

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

## S3 method for class 'ConservationProblem'
eval_boundary_summary(
  x,
  solution,
  edge_factor = rep(0.5, number_of_zones(x)),
  zones = diag(number_of_zones(x)),
  data = NULL,
  ...
)
```

Arguments

x	problem() (i.e. ConservationProblem) object.
...	not used.
solution	numeric, matrix, data.frame, Raster, Spatial, or sf::sf() object. The argument should be in the same format as the planning unit cost data in the argument to x. See the Solution format section for more information.
edge_factor	numeric proportion to scale planning unit edges (borders) that do not have any neighboring planning units. For example, an edge factor of 0.5 is commonly used to avoid overly penalizing planning units along a coastline. Note that this argument must have an element for each zone in the argument to x.
zones	matrix or Matrix object describing the clumping scheme for different zones. Each row and column corresponds to a different zone in the argument to x, and cell values indicate the relative importance of clumping planning units that are allocated to a combination of zones. Cell values along the diagonal of the matrix represent the relative importance of clumping planning units that are allocated to the same zone. Cell values must range between 1 and -1, where negative values favor solutions that spread out planning units. The default argument to zones is an identity matrix (i.e. a matrix with ones along the matrix diagonal and zeros elsewhere), so that penalties are incurred when neighboring planning units are not assigned to the same zone. If the cells along the matrix diagonal contain markedly smaller values than those found elsewhere in the matrix, then solutions are preferred that surround planning units with those allocated to different zones (i.e. greater spatial fragmentation).

data — NULL, data.frame, matrix, or Matrix object containing the boundary data. These data describe the total amount of boundary (perimeter) length for each planning unit, and the amount of boundary (perimeter) length shared between different planning units (i.e. planning units that are adjacent to each other). See the Data format section for more information.

Details

This summary statistic is equivalent to the Connectivity_Edge metric reported by the *Marxan* software (Ball *et al.* 2009). It is calculated using the same equations used to penalize solutions according to their total exposed boundary (i.e. add_boundary_penalties()). See the Examples section for examples on how differences zone arguments can be used to calculate boundaries for different combinations of zones.

Value

tibble::tibble() object containing the boundary length of the solution. It contains the following columns:

summary character description of the summary statistic. The statistic associated with the "overall" value in this column is calculated using the entire solution (including all management zones if there are multiple zones). If multiple management zones are present, then summary statistics are also provided for each zone separately (indicated using zone names).

boundary numeric exposed boundary length value. Greater values correspond to solutions with greater boundary length and, in turn, greater spatial fragmentation. Thus conservation planning exercises typically prefer solutions with smaller values.

Data format

The argument to data can be specified using the following formats. Note that boundary data must always describe symmetric relationships between planning units.

NULL boundary data are automatically calculated using the boundary_matrix() function. This argument is the default. Note that the boundary data must be supplied using one of the other formats below if the planning unit data in the argument to x do not explicitly contain spatial information (e.g. planning unit data are a data.frame or numeric class).

matrix, Matrix where rows and columns represent different planning units and the value of each cell represents the amount of shared boundary length between two different planning units. Cells that occur along the matrix diagonal represent the amount of exposed boundary associated with each planning unit that has no neighbor (e.g. these value might pertain to boundaries along a coastline).

data.frame with the columns "id1", "id2", and "boundary". The "id1" and "id2" columns contain identifiers (indices) for a pair of planning units, and the "boundary" column contains the amount of shared boundary length between these two planning units. This format follows the the standard *Marxan* format for boundary data (i.e. per the "bound.dat" file).

Solution format

Broadly speaking, the argument to `solution` must be in the same format as the planning unit data in the argument to `x`. Further details on the correct format are listed separately for each of the different planning unit data formats:

`x` **has** `numeric` **planning units** The argument to `solution` must be a `numeric` vector with each element corresponding to a different planning unit. It should have the same number of planning units as those in the argument to `x`. Additionally, any planning units missing cost (`NA`) values should also have missing (`NA`) values in the argument to `solution`.

`x` **has** `matrix` **planning units** The argument to `solution` must be a `matrix` vector with each row corresponding to a different planning unit, and each column correspond to a different management zone. It should have the same number of planning units and zones as those in the argument to `x`. Additionally, any planning units missing cost (`NA`) values for a particular zone should also have a missing (`NA`) values in the argument to `solution`.

`x` **has** `Raster` **planning units** The argument to `solution` be a `Raster` object where different grid cells (pixels) correspond to different planning units and layers correspond to a different management zones. It should have the same dimensionality (rows, columns, layers), resolution, extent, and coordinate reference system as the planning units in the argument to `x`. Additionally, any planning units missing cost (`NA`) values for a particular zone should also have missing (`NA`) values in the argument to `solution`.

`x` **has** `data.frame` **planning units** The argument to `solution` must be a `data.frame` with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if a `data.frame` object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with `sf::sf()` data). Additionally, any planning units missing cost (`NA`) values for a particular zone should also have missing (`NA`) values in the argument to `solution`.

`x` **has** `Spatial` **planning units** The argument to `solution` must be a `Spatial` object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the `Spatial` object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with `sf::sf()` data). Additionally, the argument to `solution` must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (`NA`) values for a particular zone should also have missing (`NA`) values in the argument to `solution`.

`x` **has** `sf::sf()` **planning units** The argument to `solution` must be a `sf::sf()` object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the `sf::sf()` object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example). Additionally, the argument to `solution` must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (`NA`) values for a particular zone should also have missing (`NA`) values in the argument to `solution`.

References

Ball IR, Possingham HP, and Watts M (2009) *Marxan and relatives: Software for spatial conservation prioritisation* in Spatial conservation prioritisation: Quantitative methods and computational

tools. Eds Moilanen A, Wilson KA, and Possingham HP. Oxford University Press, Oxford, UK.

See Also

problem(), summaries, add_boundary_penalties().

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
data(sim_pu_raster, sim_pu_sf, sim_features,
     sim_pu_zones_sf, sim_features_zones)

# build minimal conservation problem with raster data
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve the problem
s1 <- solve(p1)

# print solution
print(s1)

# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)

# calculate boundary associated with the solution
r1 <- eval_boundary_summary(p1, s1)
print(r1)

# build minimal conservation problem with polygon (sf) data
p2 <- problem(sim_pu_sf, sim_features, cost_column = "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve the problem
s2 <- solve(p2)

# print first six rows of the attribute table
print(head(s2))

# plot solution
plot(s2[, "solution_1"])

# calculate boundary associated with the solution
```

```
r2 <- eval_boundary_summary(p2, s2[, "solution_1"])
print(r2)

# build multi-zone conservation problem with polygon (sf) data
p3 <- problem(sim_pu_zones_sf, sim_features_zones,
              cost_column = c("cost_1", "cost_2", "cost_3")) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve the problem
s3 <- solve(p3)

# print first six rows of the attribute table
print(head(s3))

# create new column representing the zone id that each planning unit
# was allocated to in the solution
s3$solution <- category_vector(
  s3[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")])
s3$solution <- factor(s3$solution)

# plot solution
plot(s3[, "solution"])

# calculate boundary associated with the solution
# here we will use the default argument for zones which treats each
# zone as completely separate, meaning that the "overall"
# boundary is just the sum of the boundaries for each zone
r3 <- eval_boundary_summary(
  p3, s3[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")])
print(r3)

# let's calculate the overall exposed boundary across the entire
# solution, assuming that the shared boundaries between planning
# units allocated to different zones "count" just as much
# as those for planning units allocated to the same zone

# in other words, let's calculate the overall exposed boundary
# across the entire solution by "combining" all selected planning units
# together (regardless of which zone they are allocated to in the solution)
r3_combined <- eval_boundary_summary(
  p3, zones = matrix(1, ncol = 3, nrow = 3),
  s3[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")])
print(r3_combined)

# we can see that the "overall" boundary is now less than the
# sum of the individual zone boundaries, because it does not
# consider the shared boundary between two planning units allocated to
# different zones as "exposed" when performing the calculations
```

```
## End(Not run)
```

---

eval_connectivity_summary
Evaluate solution connectivity

Description

Calculate the connectivity held within a solution to a conservation planning `problem()`. This summary statistic evaluates the connectivity of a solution using pair-wise connectivity values between combinations of planning units.

Usage

```
## S4 method for signature 'ConservationProblem,ANY,ANY,matrix'
eval_connectivity_summary(x, solution, zones, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,Matrix'
eval_connectivity_summary(x, solution, zones, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,data.frame'
eval_connectivity_summary(x, solution, zones, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,dgCMatrix'
eval_connectivity_summary(x, solution, zones, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,array'
eval_connectivity_summary(x, solution, zones, data)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object.
solution	`numeric`, `matrix`, `data.frame`, `Raster`, `Spatial`, or `sf::sf()` object. The argument should be in the same format as the planning unit cost data in the argument to `x`. See the Solution format section for more information.
zones	`matrix` or `Matrix` object describing the level of connectivity between different zones. Each row and column corresponds to a different zone in the argument to `x`, and cell values indicate the level of connectivity between each combination of zones. Cell values along the diagonal of the matrix represent the level of connectivity between planning units allocated to the same zone. Cell values must lay between 1 and -1, where negative values favor solutions with weak connectivity. The default argument to `zones` is an identity matrix (i.e. a matrix with ones along the matrix diagonal and zeros elsewhere), so that planning units are only considered to be connected when they are allocated to the same zone. This argument is required when the argument to `data` is a `matrix` or `Matrix` object. If the argument to `data` is an `array` or `data.frame` with zone data, this argument must explicitly be set to `NULL` otherwise an error will be thrown.

data
: matrix, Matrix, data.frame, or array object containing connectivity data. The connectivity values correspond to the strength of connectivity between different planning units. Thus connections between planning units that are associated with higher values are more favorable in the solution. See the Data format section for more information.

Details

This summary statistic is comparable to theConnectivity_Inmetric reported by the [*Marxan* software](https://marxansolutions.org) (Ball *et al.* 2009). It is calculated using the same equations used to penalize solutions with connectivity data (i.e. [add_connectivity_penalties()]). Specifically, it is calculated as the sum of the pair-wise connectivity values in the argument todata‘, weighted by the value of the planning units in the solution.

Value

tibble::tibble() object describing the connectivity of the solution. It contains the following columns:

summary character description of the summary statistic. The statistic associated with the "overall" value in this column is calculated using the entire solution (including all management zones if there are multiple zones). If multiple management zones are present, then summary statistics are also provided for each zone separately (indicated using zone names).

connectivity numeric connectivity value. Greater values correspond to solutions associated with greater connectivity. Thus conservation planning exercises typically prefer solutions with greater values.

Solution format

Broadly speaking, the argument to solution must be in the same format as the planning unit data in the argument to x. Further details on the correct format are listed separately for each of the different planning unit data formats:

x **has** numeric **planning units** The argument to solution must be a numeric vector with each element corresponding to a different planning unit. It should have the same number of planning units as those in the argument to x. Additionally, any planning units missing cost (NA) values should also have missing (NA) values in the argument to solution.

x **has** matrix **planning units** The argument to solution must be a matrix vector with each row corresponding to a different planning unit, and each column correspond to a different management zone. It should have the same number of planning units and zones as those in the argument to x. Additionally, any planning units missing cost (NA) values for a particular zone should also have a missing (NA) values in the argument to solution.

x **has** Raster **planning units** The argument to solution be a Raster object where different grid cells (pixels) correspond to different planning units and layers correspond to a different management zones. It should have the same dimensionality (rows, columns, layers), resolution, extent, and coordinate reference system as the planning units in the argument to x. Additionally, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

x **has** data.frame **planning units** The argument to solution must be a data.frame with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if a data.frame object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with sf::sf() data). Additionally, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

x **has** Spatial **planning units** The argument to solution must be a Spatial object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the Spatial object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with sf::sf() data). Additionally, the argument to solution must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

x **has** sf::sf() **planning units** The argument to solution must be a sf::sf() object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the sf::sf() object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example). Additionally, the argument to solution must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

Data format

The argument to data can be specified using several different formats. These formats can be used to describe symmetric or asymmetric relationships between planning units.

matrix, Matrix where rows and columns represent different planning units and the value of each cell represents the strength of connectivity between two different planning units. Cells that occur along the matrix diagonal are treated as weights which indicate that planning units are more desirable in the solution. The argument to zones can be used to control the strength of connectivity between planning units in different zones. The default argument for zones is to treat planning units allocated to different zones as having zero connectivity.

data.frame containing the fields (columns) "id1", "id2", and "boundary". Here, each row denotes the connectivity between two planning units following the *Marxan* format. The data can be used to denote symmetric or asymmetric relationships between planning units. By default, input data is assumed to be symmetric unless asymmetric data is also included (e.g. if data is present for planning units 2 and 3, then the same amount of connectivity is expected for planning units 3 and 2, unless connectivity data is also provided for planning units 3 and 2). If the argument to x contains multiple zones, then the columns "zone1" and "zone2" can optionally be provided to manually specify the connectivity values between planning units when they are allocated to specific zones. If the columns "zone1" and "zone2" are present, then the argument to zones must be NULL.

array containing four-dimensions where cell values indicate the strength of connectivity between planning units when they are assigned to specific management zones. The first two dimensions (i.e. rows and columns) indicate the strength of connectivity between different planning units and the second two dimensions indicate the different management zones. Thus the `data[1,2,3,4]` indicates the strength of connectivity between planning unit 1 and planning unit 2 when planning unit 1 is assigned to zone 3 and planning unit 2 is assigned to zone 4.

References

Ball IR, Possingham HP, and Watts M (2009) *Marxan and relatives: Software for spatial conservation prioritisation* in Spatial conservation prioritisation: Quantitative methods and computational tools. Eds Moilanen A, Wilson KA, and Possingham HP. Oxford University Press, Oxford, UK.

See Also

`problem()`, summaries, `add_connectivity_penalties()`.

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
data(sim_pu_raster, sim_pu_sf, sim_features,
     sim_pu_zones_sf, sim_features_zones)

# build minimal conservation problem with raster data
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve the problem
s1 <- solve(p1)

# print solution
print(s1)

# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)

# simulate a connectivity matrix to describe the relative strength
# of connectivity between different planning units
# for brevity, we will use cost data here so that pairs
# of adjacent planning units with higher cost values will have a
# higher connectivity value
# (but see ?connectivity_matrix for more information)
cm1 <- connectivity_matrix(sim_pu_raster, sim_pu_raster)

# calculate connectivity associated with the solution
r1 <- eval_connectivity_summary(p1, s1, data = cm1)
print(r1)
```

```
# build minimal conservation problem with polygon (sf) data
p2 <- problem(sim_pu_sf, sim_features, cost_column = "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve the problem
s2 <- solve(p2)

# print first six rows of the attribute table
print(head(s2))

# plot solution
plot(s2[, "solution_1"])

# simulate connectivity matrix
# here, we will generate connectivity values randomly
# between all pairs of planning units
cm2 <- matrix(runif(nrow(sim_pu_sf) ^ 2), nrow = nrow(sim_pu_sf))

# calculate connectivity associated with the solution
r2 <- eval_connectivity_summary(p2, s2[, "solution_1"], data = cm2)
print(r2)

# build multi-zone conservation problem with polygon (sf) data
p3 <- problem(sim_pu_zones_sf, sim_features_zones,
              cost_column = c("cost_1", "cost_2", "cost_3")) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve the problem
s3 <- solve(p3)

# print first six rows of the attribute table
print(head(s3))

# create new column representing the zone id that each planning unit
# was allocated to in the solution
s3$solution <- category_vector(
  s3[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")])
s3$solution <- factor(s3$solution)

# plot solution
plot(s3[, "solution"])

# simulate connectivity matrix
# here, we will add a new column to sim_pu_zones_sf with
# randomly simulated values and create a connectivity matrix
# based on the average simulated values of adjacent planning units
```

```
sim_pu_zones_sf$con <- runif(nrow(sim_pu_zones_sf))
cm3 <- connectivity_matrix(sim_pu_zones_sf, "con")

# calculate connectivity associated with the solution
r3 <- eval_connectivity_summary(
  p3, s3[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")],
  data = cm3)
print(r3)


## End(Not run)
```

`eval_cost_summary` *Evaluate solution cost*

Description

Calculate the total cost of a solution to a conservation planning `problem()`. For example, if the planning unit cost data describe land acquisition costs (USD), then the total cost would be net cost (USD) needed to acquire all planning units selected within the solution.

Usage

```
eval_cost_summary(x, solution)

## Default S3 method:
eval_cost_summary(x, solution)

## S3 method for class 'ConservationProblem'
eval_cost_summary(x, solution)
```

Arguments

`x`	`problem()` (i.e. `ConservationProblem`) object.
`solution`	`numeric`, `matrix`, `data.frame`, `Raster`, `Spatial`, or `sf::sf()` object. The argument should be in the same format as the planning unit cost data in the argument to `x`. See the Solution format section for more information.

Details

This metric is equivalent to the `Cost` metric reported by the *Marxan* software (Ball *et al.* 2009). Specifically, the cost of a solution is defined as the sum of the cost values, supplied when creating a `problem()` object (e.g. using the `cost_column` argument), weighted by the status of each planning unit in the solution.

Value

`tibble::tibble()` object containing the solution cost. It contains the following columns:

summary `character` description of the summary statistic. The statistic associated with the `"overall"` value in this column is calculated using the entire solution (including all management zones if there are multiple zones). If multiple management zones are present, then summary statistics are also provided for each zone separately (indicated using zone names).

cost `numeric` cost value. Greater values correspond to solutions that are more costly to implement. Thus conservation planning exercises typically prefer solutions with smaller values, because they are cheaper to implement (assuming all other relevant factors, such as feature representation, are equal).

Solution format

Broadly speaking, the argument to `solution` must be in the same format as the planning unit data in the argument to `x`. Further details on the correct format are listed separately for each of the different planning unit data formats:

`x` **has** `numeric` **planning units** The argument to `solution` must be a `numeric` vector with each element corresponding to a different planning unit. It should have the same number of planning units as those in the argument to `x`. Additionally, any planning units missing cost (`NA`) values should also have missing (`NA`) values in the argument to `solution`.

`x` **has** `matrix` **planning units** The argument to `solution` must be a `matrix` vector with each row corresponding to a different planning unit, and each column correspond to a different management zone. It should have the same number of planning units and zones as those in the argument to `x`. Additionally, any planning units missing cost (`NA`) values for a particular zone should also have a missing (`NA`) values in the argument to `solution`.

`x` **has** `Raster` **planning units** The argument to `solution` be a `Raster` object where different grid cells (pixels) correspond to different planning units and layers correspond to a different management zones. It should have the same dimensionality (rows, columns, layers), resolution, extent, and coordinate reference system as the planning units in the argument to `x`. Additionally, any planning units missing cost (`NA`) values for a particular zone should also have missing (`NA`) values in the argument to `solution`.

`x` **has** `data.frame` **planning units** The argument to `solution` must be a `data.frame` with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if a `data.frame` object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with `sf::sf()` data). Additionally, any planning units missing cost (`NA`) values for a particular zone should also have missing (`NA`) values in the argument to `solution`.

`x` **has** `Spatial` **planning units** The argument to `solution` must be a `Spatial` object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the `Spatial` object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with `sf::sf()` data). Additionally, the argument to `solution` must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (`NA`) values for a particular zone should also have missing (`NA`) values in the argument to `solution`.

`x` **has `sf::sf()` planning units** The argument to `solution` must be a `sf::sf()` object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the `sf::sf()` object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example). Additionally, the argument to `solution` must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (`NA`) values for a particular zone should also have missing (`NA`) values in the argument to `solution`.

References

Ball IR, Possingham HP, and Watts M (2009) *Marxan and relatives: Software for spatial conservation prioritisation* in Spatial conservation prioritisation: Quantitative methods and computational tools. Eds Moilanen A, Wilson KA, and Possingham HP. Oxford University Press, Oxford, UK.

See Also

`problem()`, summaries.

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
data(sim_pu_raster, sim_pu_sf, sim_features,
     sim_pu_zones_sf, sim_features_zones)

# build minimal conservation problem with raster data
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve the problem
s1 <- solve(p1)

# print solution
print(s1)

# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)

# calculate cost of the solution
r1 <- eval_cost_summary(p1, s1)
print(r1)

# build minimal conservation problem with polygon (sf) data
p2 <- problem(sim_pu_sf, sim_features, cost_column = "cost") %>%
      add_min_set_objective() %>%
```

```
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve the problem
s2 <- solve(p2)

# plot solution
plot(s2[, "solution_1"])

# print first six rows of the attribute table
print(head(s2))

# calculate cost of the solution
r2 <- eval_cost_summary(p2, s2[, "solution_1"])
print(r2)

# manually calculate cost of the solution
r2_manual <- sum(s2$solution * sim_pu_sf$cost, na.rm = TRUE)
print(r2_manual)

# build multi-zone conservation problem with polygon (sf) data
p3 <- problem(sim_pu_zones_sf, sim_features_zones,
              cost_column = c("cost_1", "cost_2", "cost_3")) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve the problem
s3 <- solve(p3)

# print first six rows of the attribute table
print(head(s3))

# create new column representing the zone id that each planning unit
# was allocated to in the solution
s3$solution <- category_vector(
  s3[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")])
s3$solution <- factor(s3$solution)

# plot solution
plot(s3[, "solution"])

# calculate cost of the solution
r3 <- eval_cost_summary(
  p3, s3[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")])
print(r3)

## End(Not run)
```

eval_feature_representation_summary
Evaluate feature representation

Description

Calculate how well features are represented by a solution to a conservation planning `problem()`. These summary statistics are reported for each and every feature, and each and every zone, within a conservation planning problem.

Usage

```
eval_feature_representation_summary(x, solution)

## S4 method for signature 'ConservationProblem,numeric'
eval_feature_representation_summary(x, solution)

## S4 method for signature 'ConservationProblem,matrix'
eval_feature_representation_summary(x, solution)

## S4 method for signature 'ConservationProblem,data.frame'
eval_feature_representation_summary(x, solution)

## S4 method for signature 'ConservationProblem,Spatial'
eval_feature_representation_summary(x, solution)

## S4 method for signature 'ConservationProblem,sf'
eval_feature_representation_summary(x, solution)

## S4 method for signature 'ConservationProblem,Raster'
eval_feature_representation_summary(x, solution)
```

Arguments

`x`	`problem()` (i.e. `ConservationProblem`) object.
`solution`	`numeric`, `matrix`, `data.frame`, `Raster`, `Spatial`, or `sf::sf()` object. The argument should be in the same format as the planning unit cost data in the argument to `x`. See the Solution format section for more information.

Value

`tibble::tibble()` object describing feature representation. Here, each row describes a specific summary statistic (e.g. different management zone) for a specific feature. It contains the following columns:

summary `character` description of the summary statistic. The statistic associated with the `"overall"` value in this column is calculated using the entire solution (including all management zones if there are multiple zones). If multiple management zones are present, then summary statistics are also provided for each zone separately (indicated using zone names).

feature `character` name of the feature.

total_amount `numeric` total amount of each feature available in the entire conservation planning problem (not just planning units selected within the solution). It is calculated as the sum of the feature data, supplied when creating a `problem()` object (e.g. presence/absence values).

absolute_held `numeric` total amount of each feature secured within the solution. It is calculated as the sum of the feature data, supplied when creating a `problem()` object (e.g. presence/absence values), weighted by the status of each planning unit in the solution (e.g. selected or not for prioritization).

relative_held `numeric` proportion of each feature secured within the solution. It is calculated by dividing values in the `"absolute_held"` column by those in the `"total_amount"` column.

Solution format

Broadly speaking, the argument to `solution` must be in the same format as the planning unit data in the argument to `x`. Further details on the correct format are listed separately for each of the different planning unit data formats:

`x` **has** `numeric` **planning units** The argument to `solution` must be a `numeric` vector with each element corresponding to a different planning unit. It should have the same number of planning units as those in the argument to `x`. Additionally, any planning units missing cost (`NA`) values should also have missing (`NA`) values in the argument to `solution`.

`x` **has** `matrix` **planning units** The argument to `solution` must be a `matrix` vector with each row corresponding to a different planning unit, and each column correspond to a different management zone. It should have the same number of planning units and zones as those in the argument to `x`. Additionally, any planning units missing cost (`NA`) values for a particular zone should also have a missing (`NA`) values in the argument to `solution`.

`x` **has** `Raster` **planning units** The argument to `solution` be a `Raster` object where different grid cells (pixels) correspond to different planning units and layers correspond to a different management zones. It should have the same dimensionality (rows, columns, layers), resolution, extent, and coordinate reference system as the planning units in the argument to `x`. Additionally, any planning units missing cost (`NA`) values for a particular zone should also have missing (`NA`) values in the argument to `solution`.

`x` **has** `data.frame` **planning units** The argument to `solution` must be a `data.frame` with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if a `data.frame` object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with `sf::sf()` data). Additionally, any planning units missing cost (`NA`) values for a particular zone should also have missing (`NA`) values in the argument to `solution`.

`x` **has** `Spatial` **planning units** The argument to `solution` must be a `Spatial` object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the `Spatial` object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with `sf::sf()` data). Additionally, the argument to `solution` must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

x **has sf::sf() planning units** The argument to solution must be a sf::sf() object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the sf::sf() object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example). Additionally, the argument to solution must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

See Also

problem(), summaries.

Examples

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

# load data
data(sim_pu_raster, sim_pu_polygons, sim_pu_zones_sf, sim_features,
     sim_pu_zones_stack, sim_features_zones)

# create a simple conservation planning dataset so we can see exactly
# how feature representation is calculated
pu <- data.frame(id = seq_len(10), cost = c(0.2, NA, runif(8)),
                 spp1 = runif(10), spp2 = c(rpois(9, 4), NA))

# create problem
p1 <- problem(pu, c("spp1", "spp2"), cost_column = "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create a solution
# specifically, a data.frame with a single column that contains
# binary values indicating if each planning units was selected or not
s1 <- data.frame(s = c(1, NA, rep(c(1, 0), 4)))
print(s1)

# calculate feature representation
r1 <- eval_feature_representation_summary(p1, s1)
print(r1)

# let's verify that feature representation calculations are correct
# by manually performing the calculations and compare the results with r1
## calculate total amount for each feature
print(setNames(
  c(sum(pu$spp1, na.rm = TRUE),
    sum(pu$spp2, na.rm = TRUE)),
```

```
  c("spp1", "spp2")))

## calculate absolute amount held for each feature
print(setNames(
  c(sum(pu$spp1 * s1$s, na.rm = TRUE),
    sum(pu$spp2 * s1$s, na.rm = TRUE)),
  c("spp1", "spp2")))

## calculate relative amount held for each feature
print(setNames(
  c(sum(pu$spp1 * s1$s, na.rm = TRUE) / sum(pu$spp1, na.rm = TRUE),
    sum(pu$spp2 * s1$s, na.rm = TRUE) / sum(pu$spp2, na.rm = TRUE)),
  c("spp1", "spp2")))

## Not run:
# solve the problem using an exact algorithm solver
s1_2 <- solve(p1)
print(s1_2)

# calculate feature representation in this solution
r1_2 <- eval_feature_representation_summary(
  p1, s1_2[, "solution_1", drop = FALSE])
print(r1_2)

# build minimal conservation problem with raster data
p2 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve the problem
s2 <- solve(p2)

# print solution
print(s2)

# calculate feature representation in the solution
r2 <- eval_feature_representation_summary(p2, s2)
print(r2)

# plot solution
plot(s2, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
# build minimal conservation problem with polygon (Spatial) data
p3 <- problem(sim_pu_polygons, sim_features, cost_column = "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve the problem
```

```
s3 <- solve(p3)

# print first six rows of the attribute table
print(head(s3))

# calculate feature representation in the solution
r3 <- eval_feature_representation_summary(p3, s3[, "solution_1"])
print(r3)

# plot solution
spplot(s3, zcol = "solution_1", main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
# build multi-zone conservation problem with raster data
p4 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve the problem
s4 <- solve(p4)

# print solution
print(s4)

# calculate feature representation in the solution
r4 <- eval_feature_representation_summary(p4, s4)
print(r4)

# plot solution
plot(category_layer(s4), main = "solution", axes = FALSE, box = FALSE)

## End(Not run)

# build multi-zone conservation problem with polygon (sf) data
p5 <- problem(sim_pu_zones_sf, sim_features_zones,
              cost_column = c("cost_1", "cost_2", "cost_3")) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve the problem
s5 <- solve(p5)

# print first six rows of the attribute table
print(head(s5))

# calculate feature representation in the solution
r5 <- eval_feature_representation_summary(
```

```
  p5, s5[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")])
print(r5)

# create new column representing the zone id that each planning unit
# was allocated to in the solution
s5$solution <- category_vector(
  s5[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")])
s5$solution <- factor(s5$solution)

# plot solution
plot(s5[, "solution"])

## End(Not run)
```

`eval_ferrier_importance` *Evaluate solution importance using Ferrier scores*

Description

Calculate importance scores for planning units selected in a solution following Ferrier *et al.* (2000). **Please note that the mathematical formulation for computing these scores needs verification, and so this functionality should be considered experimental at this point in time.**

Usage

```
eval_ferrier_importance(x, solution)

## S4 method for signature 'ConservationProblem,numeric'
eval_ferrier_importance(x, solution)

## S4 method for signature 'ConservationProblem,matrix'
eval_ferrier_importance(x, solution)

## S4 method for signature 'ConservationProblem,data.frame'
eval_ferrier_importance(x, solution)

## S4 method for signature 'ConservationProblem,Spatial'
eval_ferrier_importance(x, solution)

## S4 method for signature 'ConservationProblem,sf'
eval_ferrier_importance(x, solution)

## S4 method for signature 'ConservationProblem,Raster'
eval_ferrier_importance(x, solution)
```

Arguments

x: problem() (i.e. ConservationProblem) object.

solution: numeric, matrix, data.frame, Raster, Spatial, or sf::sf() object. The argument should be in the same format as the planning unit cost data in the argument to x. See the Solution format section for more information.

Details

Importance scores are reported separately for each feature within each planning unit. Additionally, a total importance score is also calculated as the sum of the scores for each feature. Note that this function only works for problems with a minimum set objective and a single zone. It will throw an error for other types of problems that do not meet this specification.

Value

A matrix, tibble::tibble(), RasterLayer, or Spatial object containing the scores for each planning unit selected in the solution. Specifically, the returned object is in the same format (except if the planning units are a numeric vector) as the planning unit data in the argument to x.

Solution format

Broadly speaking, the argument to solution must be in the same format as the planning unit data in the argument to x. Further details on the correct format are listed separately for each of the different planning unit data formats:

x **has** numeric **planning units** The argument to solution must be a numeric vector with each element corresponding to a different planning unit. It should have the same number of planning units as those in the argument to x. Additionally, any planning units missing cost (NA) values should also have missing (NA) values in the argument to solution.

x **has** matrix **planning units** The argument to solution must be a matrix vector with each row corresponding to a different planning unit, and each column correspond to a different management zone. It should have the same number of planning units and zones as those in the argument to x. Additionally, any planning units missing cost (NA) values for a particular zone should also have a missing (NA) values in the argument to solution.

x **has** Raster **planning units** The argument to solution be a Raster object where different grid cells (pixels) correspond to different planning units and layers correspond to a different management zones. It should have the same dimensionality (rows, columns, layers), resolution, extent, and coordinate reference system as the planning units in the argument to x. Additionally, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

x **has** data.frame **planning units** The argument to solution must be a data.frame with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if a data.frame object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with sf::sf() data). Additionally, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

x **has Spatial planning units** The argument to solution must be a Spatial object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the Spatial object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with sf::sf() data). Additionally, the argument to solution must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

x **has sf::sf() planning units** The argument to solution must be a sf::sf() object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the sf::sf() object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example). Additionally, the argument to solution must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

References

Ferrier S, Pressey RL, and Barrett TW (2000) A new predictor of the irreplaceability of areas for achieving a conservation goal, its application to real-world planning, and a research agenda for further refinement. *Biological Conservation*, 93: 303–325.

See Also

importance.

Examples

```
# seed seed for reproducibility
set.seed(600)

# load data
data(sim_pu_raster, sim_features)

# create minimal problem with binary decisions
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(gap = 0, verbose = FALSE)
## Not run:
# solve problem
s1 <- solve(p1)

# print solution
print(s1)

# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)
```

```
# calculate importance scores using Ferrier et al. 2000 method
fs1 <- eval_ferrier_importance(p1, s1)

# print importance scores,
# each planning unit has an importance score for each feature
# (as indicated by the column names) and each planning unit also
# has an overall total importance score (in the "total" column)
print(fs1)

# plot total importance scores
plot(fs1, main = "Ferrier scores", axes = FALSE, box = FALSE)

# create minimal problem with polygon (sf) planning units
p2 <- problem(sim_pu_sf, sim_features, cost_column = "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.05) %>%
      add_binary_decisions() %>%
      add_default_solver(gap = 0, verbose = FALSE)

# solve problem
s2 <- solve(p2)

# print solution
print(s2)

# plot solution
plot(s2[, "solution_1"], main = "solution")

# calculate importance scores
fs2 <- eval_ferrier_importance(p2, s2[, "solution_1"])

# plot importance scores
plot(fs2, main = "Ferrier scores")


## End(Not run)
```

`eval_n_summary` *Evaluate number of planning units selected.*

Description

Calculate the number of planning units selected within a solution to a conservation planning `problem()`.

Usage

```
eval_n_summary(x, solution)

## Default S3 method:
```

```
eval_n_summary(x, solution)

## S3 method for class 'ConservationProblem'
eval_n_summary(x, solution)
```

Arguments

x	problem() (i.e. ConservationProblem) object.
solution	numeric, matrix, data.frame, Raster, Spatial, or sf::sf() object. The argument should be in the same format as the planning unit cost data in the argument to x. See the Solution format section for more information.

Details

This summary statistic is calculated as the sum of the values in the solution. As a consequence, this metric can produce a non-integer value (e.g. 4.3) for solutions containing proportion values (e.g. generated by solving a problem() built using the add_proportion_decisions() function).

Value

tibble::tibble() object containing the number of planning units selected within a solution. It contains the following columns:

summary character description of the summary statistic. The statistic associated with the "overall" value in this column is calculated using the entire solution (including all management zones if there are multiple zones). If multiple management zones are present, then summary statistics are also provided for each zone separately (indicated using zone names).

n numeric number of selected planning units.

Solution format

Broadly speaking, the argument to solution must be in the same format as the planning unit data in the argument to x. Further details on the correct format are listed separately for each of the different planning unit data formats:

x **has** numeric **planning units** The argument to solution must be a numeric vector with each element corresponding to a different planning unit. It should have the same number of planning units as those in the argument to x. Additionally, any planning units missing cost (NA) values should also have missing (NA) values in the argument to solution.

x **has** matrix **planning units** The argument to solution must be a matrix vector with each row corresponding to a different planning unit, and each column correspond to a different management zone. It should have the same number of planning units and zones as those in the argument to x. Additionally, any planning units missing cost (NA) values for a particular zone should also have a missing (NA) values in the argument to solution.

x **has** Raster **planning units** The argument to solution be a Raster object where different grid cells (pixels) correspond to different planning units and layers correspond to a different management zones. It should have the same dimensionality (rows, columns, layers), resolution, extent, and coordinate reference system as the planning units in the argument to x. Additionally, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

x **has** data.frame **planning units** The argument to solution must be a data.frame with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if a data.frame object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with sf::sf() data). Additionally, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

x **has** Spatial **planning units** The argument to solution must be a Spatial object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the Spatial object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with sf::sf() data). Additionally, the argument to solution must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

x **has** sf::sf() **planning units** The argument to solution must be a sf::sf() object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the sf::sf() object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example). Additionally, the argument to solution must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

See Also

problem(), summaries.

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
data(sim_pu_raster, sim_pu_sf, sim_features,
     sim_pu_zones_sf, sim_features_zones)

# build minimal conservation problem with raster data
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve the problem
```

```
s1 <- solve(p1)

# print solution
print(s1)

# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)

# calculate number of selected planning units within solution
r1 <- eval_n_summary(p1, s1)
print(r1)

# build minimal conservation problem with polygon (sf) data
p2 <- problem(sim_pu_sf, sim_features, cost_column = "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve the problem
s2 <- solve(p2)

# plot solution
plot(s2[, "solution_1"])

# print first six rows of the attribute table
print(head(s2))

# calculate number of selected planning units within solution
r2 <- eval_n_summary(p2, s2[, "solution_1"])
print(r2)

# manually calculate number of selected planning units
r2_manual <- sum(s2$solution, na.rm = TRUE)
print(r2_manual)

# build multi-zone conservation problem with polygon (sf) data
p3 <- problem(sim_pu_zones_sf, sim_features_zones,
              cost_column = c("cost_1", "cost_2", "cost_3")) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve the problem
s3 <- solve(p3)

# print first six rows of the attribute table
print(head(s3))

# create new column representing the zone id that each planning unit
# was allocated to in the solution
```

```
s3$solution <- category_vector(
  s3[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")])
s3$solution <- factor(s3$solution)

# plot solution
plot(s3[, "solution"])

# calculate number of selected planning units within solution
r3 <- eval_n_summary(
  p3, s3[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")])
print(r3)

## End(Not run)
```

`eval_rare_richness_importance`

Evaluate solution importance using rarity weighted richness scores

---

Description

Calculate importance scores for planning units selected in a solution using rarity weighted richness scores (based on Williams *et al.* 1996). This method is only recommended for large-scaled conservation planning exercises (i.e. more than 100,000 planning units) where importance scores cannot be calculated using other methods in a feasible period of time. This is because rarity weighted richness scores cannot (i) account for the cost of different planning units, (ii) account for multiple management zones, and (iii) identify truly irreplaceable planning units— unlike the replacement cost metric which does not suffer any of these limitations.

Usage

```
eval_rare_richness_importance(x, solution, ...)

## S4 method for signature 'ConservationProblem,numeric'
eval_rare_richness_importance(x, solution, rescale, ...)

## S4 method for signature 'ConservationProblem,matrix'
eval_rare_richness_importance(x, solution, rescale, ...)

## S4 method for signature 'ConservationProblem,data.frame'
eval_rare_richness_importance(x, solution, rescale, ...)

## S4 method for signature 'ConservationProblem,Spatial'
eval_rare_richness_importance(x, solution, rescale, ...)

## S4 method for signature 'ConservationProblem,sf'
eval_rare_richness_importance(x, solution, rescale, ...)

## S4 method for signature 'ConservationProblem,Raster'
eval_rare_richness_importance(x, solution, rescale, ...)
```

Arguments

x	problem() (i.e. ConservationProblem) object.
solution	numeric, matrix, data.frame, Raster, Spatial, or sf::sf() object. The argument should be in the same format as the planning unit cost data in the argument to x. See the Solution format section for more information.
...	not used.
rescale	logical flag indicating if replacement cost values—excepting infinite (Inf) and zero values—should be rescaled to range between 0.01 and 1. Defaults to TRUE.

Details

Rarity weighted richness scores are calculated using the following terms. Let I denote the set of planning units (indexed by i), let J denote the set of conservation features (indexed by j), let r_{ij} denote the amount of feature j associated with planning unit i, and let m_j denote the maximum value of feature j in r_{ij} in all planning units $i \in I$. To calculate the rarity weighted richness (*RWR*) for planning unit k:

$$RWR_k = \sum_{j}^{J} \frac{\frac{r_{ik}}{m_j}}{\sum_{i}^{I} r_{ij}}$$

Value

A numeric, matrix, data.frame RasterLayer, Spatial, or sf::sf() object containing the importance scores for each planning unit in the solution. Specifically, the returned object is in the same format as the planning unit data in the argument to x.

Solution format

Broadly speaking, the argument to solution must be in the same format as the planning unit data in the argument to x. Further details on the correct format are listed separately for each of the different planning unit data formats:

x **has** numeric **planning units** The argument to solution must be a numeric vector with each element corresponding to a different planning unit. It should have the same number of planning units as those in the argument to x. Additionally, any planning units missing cost (NA) values should also have missing (NA) values in the argument to solution.

x **has** matrix **planning units** The argument to solution must be a matrix vector with each row corresponding to a different planning unit, and each column correspond to a different management zone. It should have the same number of planning units and zones as those in the argument to x. Additionally, any planning units missing cost (NA) values for a particular zone should also have a missing (NA) values in the argument to solution.

x **has** Raster **planning units** The argument to solution be a Raster object where different grid cells (pixels) correspond to different planning units and layers correspond to a different management zones. It should have the same dimensionality (rows, columns, layers), resolution, extent, and coordinate reference system as the planning units in the argument to x. Additionally, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

x **has** `data.frame` **planning units** The argument to `solution` must be a `data.frame` with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if a `data.frame` object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with `sf::sf()` data). Additionally, any planning units missing cost (`NA`) values for a particular zone should also have missing (`NA`) values in the argument to `solution`.

x **has** `Spatial` **planning units** The argument to `solution` must be a `Spatial` object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the `Spatial` object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with `sf::sf()` data). Additionally, the argument to `solution` must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (`NA`) values for a particular zone should also have missing (`NA`) values in the argument to `solution`.

x **has** `sf::sf()` **planning units** The argument to `solution` must be a `sf::sf()` object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the `sf::sf()` object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example). Additionally, the argument to `solution` must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (`NA`) values for a particular zone should also have missing (`NA`) values in the argument to `solution`.

References

Williams P, Gibbons D, Margules C, Rebelo A, Humphries C, and Pressey RL (1996) A comparison of richness hotspots, rarity hotspots and complementary areas for conserving diversity using British birds. *Conservation Biology*, 10: 155–174.

See Also

importance.

Examples

```
## Not run:
# seed seed for reproducibility
set.seed(600)

# load data
data(sim_pu_raster, sim_pu_sf, sim_features)

# create minimal problem with raster planning units
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(gap = 0, verbose = FALSE)
```

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

# print solution
print(s1)

# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)

# calculate importance scores
rwr1 <- eval_rare_richness_importance(p1, s1)

# print importance scores
print(rwr1)

# plot importance scores
plot(rwr1, main = "rarity weighted richness", axes = FALSE, box = FALSE)

# create minimal problem with polygon (sf) planning units
p2 <- problem(sim_pu_sf, sim_features, cost_column = "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.05) %>%
      add_binary_decisions() %>%
      add_default_solver(gap = 0, verbose = FALSE)

# solve problem
s2 <- solve(p2)

# print solution
print(s2)

# plot solution
plot(s2[, "solution_1"], main = "solution")

# calculate importance scores
rwr2 <- eval_rare_richness_importance(p2, s2[, "solution_1"])

# plot importance scores
plot(rwr2, main = "rarity weighted richness")

## End(Not run)
```

`eval_replacement_importance`

Evaluate solution importance using replacement cost scores

Description

Calculate importance scores for planning units selected in a solution based on the replacement cost method (Cabeza and Moilanen 2006).

Usage

```
eval_replacement_importance(x, solution, ...)

## S4 method for signature 'ConservationProblem,numeric'
eval_replacement_importance(x, solution, rescale, run_checks, force, threads, ...)

## S4 method for signature 'ConservationProblem,matrix'
eval_replacement_importance(x, solution, rescale, run_checks, force, threads, ...)

## S4 method for signature 'ConservationProblem,data.frame'
eval_replacement_importance(x, solution, rescale, run_checks, force, threads, ...)

## S4 method for signature 'ConservationProblem,Spatial'
eval_replacement_importance(x, solution, rescale, run_checks, force, threads, ...)

## S4 method for signature 'ConservationProblem,sf'
eval_replacement_importance(x, solution, rescale, run_checks, force, threads, ...)

## S4 method for signature 'ConservationProblem,Raster'
eval_replacement_importance(x, solution, rescale, run_checks, force, threads, ...)
```

Arguments

`x`	`problem()` (i.e. `ConservationProblem`) object.
`solution`	`numeric`, `matrix`, `data.frame`, `Raster`, `Spatial`, or `sf::sf()` object. The argument should be in the same format as the planning unit cost data in the argument to `x`. See the Solution format section for more information.
`...`	not used.
`rescale`	`logical` flag indicating if replacement cost values—excepting infinite (`Inf`) and zero values—should be rescaled to range between 0.01 and 1. Defaults to `TRUE`.
`run_checks`	`logical` flag indicating whether presolve checks should be run prior solving the problem. These checks are performed using the `presolve_check()` function. Defaults to `TRUE`. Skipping these checks may reduce run time for large problems.
`force`	`logical` flag indicating if an attempt to should be made to solve the problem even if potential issues were detected during the presolve checks. Defaults to `FALSE`.
`threads`	`integer` number of threads to use for the optimization algorithm. Defaults to 1 such that only a single thread is used.

Details

This function implements a modified version of the replacement cost method (Cabeza and Moilanen 2006). Specifically, the score for each planning unit is calculated as the difference in the objective value of a solution when each planning unit is locked out and the optimization processes rerun with all other selected planning units locked in. In other words, the replacement cost metric corresponds to change in solution quality incurred if a given planning unit cannot be acquired when implementing the solution and the next best planning unit (or set of planning units) will need to be considered instead. Thus planning units with a higher score are more important (and irreplaceable). For example, when using the minimum set objective function (`add_min_set_objective()`), the replacement cost scores correspond to the additional costs needed to meet targets when each planning unit is locked out. When using the maximum utility objective function (`add_max_utility_objective()`, the replacement cost scores correspond to the reduction in the utility when each planning unit is locked out. Infinite values mean that no feasible solution exists when planning units are locked out—they are absolutely essential for obtaining a solution (e.g. they contain rare species that are not found in any other planning units or were locked in). Zeros values mean that planning units can swapped with other planning units and this will have no effect on the performance of the solution at all (e.g. because they were only selected due to spatial fragmentation penalties).

These calculations can take a long time to complete for large or complex conservation planning problems. As such, we using this method for small or moderate-sized conservation planning problems (e.g. < 30,000 planning units). To reduce run time, we recommend calculating these scores without additional penalties (e.g. `add_boundary_penalties()`) or spatial constraints (e.g. `add_contiguity_constraints()`). To further reduce run time, we recommend using proportion-type decisions when calculating the scores (see below for an example).

Value

A `numeric`, `matrix`, `data.frame` `RasterLayer`, `Spatial`, or `sf::sf()` object containing the importance scores for each planning unit in the solution. Specifically, the returned object is in the same format as the planning unit data in the argument to `x`.

Solution format

Broadly speaking, the argument to `solution` must be in the same format as the planning unit data in the argument to `x`. Further details on the correct format are listed separately for each of the different planning unit data formats:

`x` **has** `numeric` **planning units** The argument to `solution` must be a `numeric` vector with each element corresponding to a different planning unit. It should have the same number of planning units as those in the argument to `x`. Additionally, any planning units missing cost (`NA`) values should also have missing (`NA`) values in the argument to `solution`.

`x` **has** `matrix` **planning units** The argument to `solution` must be a `matrix` vector with each row corresponding to a different planning unit, and each column correspond to a different management zone. It should have the same number of planning units and zones as those in the argument to `x`. Additionally, any planning units missing cost (`NA`) values for a particular zone should also have a missing (`NA`) values in the argument to `solution`.

`x` **has** `Raster` **planning units** The argument to `solution` be a `Raster` object where different grid cells (pixels) correspond to different planning units and layers correspond to a different management zones. It should have the same dimensionality (rows, columns, layers), resolution, extent, and coordinate reference system as the planning units in the argument to `x`. Additionally, any planning units missing cost (`NA`) values for a particular zone should also have missing (`NA`) values in the argument to `solution`.

`x` **has** `data.frame` **planning units** The argument to `solution` must be a `data.frame` with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if a `data.frame` object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with `sf::sf()` data). Additionally, any planning units missing cost (`NA`) values for a particular zone should also have missing (`NA`) values in the argument to `solution`.

`x` **has `Spatial` planning units** The argument to `solution` must be a `Spatial` object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the `Spatial` object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with `sf::sf()` data). Additionally, the argument to `solution` must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (`NA`) values for a particular zone should also have missing (`NA`) values in the argument to `solution`.

`x` **has `sf::sf()` planning units** The argument to `solution` must be a `sf::sf()` object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the `sf::sf()` object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example). Additionally, the argument to `solution` must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (`NA`) values for a particular zone should also have missing (`NA`) values in the argument to `solution`.

References

Cabeza M and Moilanen A (2006) Replacement cost: A practical measure of site value for cost-effective reserve planning. *Biological Conservation*, 132: 336–342.

See Also

importance.

Examples

```
## Not run:
# seed seed for reproducibility
set.seed(600)

# load data
data(sim_pu_raster, sim_features, sim_pu_zones_stack, sim_features_zones)

# create minimal problem with binary decisions
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(gap = 0, verbose = FALSE)
# solve problem
s1 <- solve(p1)

# print solution
print(s1)
```

```
# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)

# calculate importance scores
rc1 <- eval_replacement_importance(p1, s1)

# print importance scores
print(rc1)

# plot importance scores
plot(rc1, main = "replacement cost", axes = FALSE, box = FALSE)

# since replacement cost scores can take a long time to calculate with
# binary decisions, we can calculate them using proportion-type
# decision variables. Note we are still calculating the scores for our
# previous solution (s1), we are just using a different optimization
# problem when calculating the scores.
p2 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_proportion_decisions() %>%
      add_default_solver(gap = 0, verbose = FALSE)

# calculate importance scores using proportion type decisions
rc2 <- eval_replacement_importance(p2, s1)

# print importance scores based on proportion type decisions
print(rc2)

# plot importance scores based on proportion type decisions
# we can see that the importance values in rc1 and rc2 are similar,
# and this confirms that the proportion type decisions are a good
# approximation
plot(rc2, main = "replacement cost", axes = FALSE, box = FALSE)

# create minimal problem with polygon (sf) planning units
p3 <- problem(sim_pu_sf, sim_features, cost_column = "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.05) %>%
      add_binary_decisions() %>%
      add_default_solver(gap = 0, verbose = FALSE)

# solve problem
s3 <- solve(p3)

# print solution
print(s3)

# plot solution
plot(s3[, "solution_1"], main = "solution")

# calculate importance scores
rc3 <- eval_rare_richness_importance(p3, s3[, "solution_1"])
```

```
# plot importance scores
plot(rc3, main = "replacement cost")

# build multi-zone conservation problem with raster data
p4 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(gap = 0, verbose = FALSE)

# solve the problem
s4 <- solve(p4)

# print solution
print(s4)

# plot solution
# each panel corresponds to a different zone, and data show the
# status of each planning unit in a given zone
plot(s4, main = paste0("zone ", seq_len(nlayers(s4))), axes = FALSE,
     box = FALSE)

# calculate importance scores
rc4 <- eval_replacement_importance(p4, s4)

# plot importance
# each panel corresponds to a different zone, and data show the
# importance of each planning unit in a given zone
plot(rc4, main = paste0("zone ", seq_len(nlayers(s4))), axes = FALSE,
     box = FALSE)

## End(Not run)
```

`eval_target_coverage_summary`

Evaluate target coverage

Description

Calculate how well feature representation targets are met by a solution to a conservation planning `problem()`. It is useful for understanding if features are adequately represented by a solution. Note that this function can only be used with problems that contain targets.

Usage

```
eval_target_coverage_summary(x, solution, include_zone, include_sense)
```

```
## Default S3 method:
eval_target_coverage_summary(x, solution, include_zone, include_sense)

## S3 method for class 'ConservationProblem'
eval_target_coverage_summary(
  x,
  solution,
  include_zone = number_of_zones(x) > 1,
  include_sense = number_of_zones(x) > 1
)
```

Arguments

x	problem() (i.e. ConservationProblem) object.
solution	numeric, matrix, data.frame, Raster, Spatial, or sf::sf() object. The argument should be in the same format as the planning unit cost data in the argument to x. See the Solution format section for more information.
include_zone	logical include the zone column in the output? Defaults to TRUE for problems that contain multiple zones.
include_sense	logical include the sense column in the output? Defaults to TRUE for problems that contain multiple zones.

Value

tibble::tibble() object. Here, each row describes information for a different target. It contains the following columns:

feature character name of the feature associated with each target.

zone list of character zone names associated with each target. This column is in a list-column format because a single target can correspond to multiple zones (see add_manual_targets() for details and examples). For an example of converting the list-column format to a standard character column format, please see the Examples section. This column is only included if the argument to include_zones is TRUE.

sense character sense associated with each target. Sense values specify the nature of the target. Typically (e.g. when using the add_absolute_targets() or add_relative_targets() functions), targets are specified using sense values indicating that the total amount of a feature held within a solution (ideally) be greater than or equal to a threshold amount (i.e. a sense value of ">="). Additionally, targets (i.e. using the add_manual_targets() function) can also be specified using sense values indicating that the total amount of a feature held within a solution must be equal to a threshold amount (i.e. a sense value of "=") or smaller than or equal to a threshold amount (i.e. a sense value of "<="). This column is only included if the argument to include_sense is TRUE.

total_amount numeric total amount of the feature available across the entire conservation planning problem for meeting each target (not just planning units selected within the solution). For problems involving a single zone, this column is calculated as the sum of all of the values for a given feature (similar to values in the total_amount column produced by the eval_feature_representation_summary() function). For problems involving multiple zones, this column is calculated as the sum of the values for the feature associated with target (per the "feature" column), across the zones associated with the target (per the "zone" column).

absolute_target numeric total threshold amount associated with each target.

absolute_held numeric total amount held within the solution for the feature and (if relevant) zones associated with each target (per the "feature" and "zone" columns, respectively). This column is calculated as the sum of the feature data, supplied when creating a problem() object (e.g. presence/absence values), weighted by the status of each planning unit in the solution (e.g. selected or not for prioritization).

absolute_shortfall numeric total amount by which the solution fails to meet each target. This column is calculated as the difference between the total amount held within the solution for the feature and (if relevant) zones associated with the target (i.e. "absolute_held" column) and the target total threshold amount (i.e. "absolute_target" column), with values set to zero depending on the sense specified for the target (e.g. if the target sense is >= then the difference is set to zero if the value in the "absolute_held" is smaller than that in the "absolute_target" column).

relative_target numeric proportion threshold amount associated with each target. This column is calculated by dividing the total threshold amount associated with each target (i.e. "absolute_target" column) by the total amount associated with each target (i.e. "total_amount" column).

relative_held numeric proportion held within the solution for the feature and (if relevant) zones associated with each target (per the "feature" and "zone" columns, respectively). This column is calculated by dividing the total amount held for each target (i.e. "absolute_held" column) by the total amount for with each target (i.e. "total_amount" column).

relative_shortfall numeric proportion by which the solution fails to meet each target. This column is calculated by dividing the total shortfall for each target (i.e. "absolute_shortfall" column) by the total amount for each target (i.e. "total_amount" column).

met logical indicating if each target is met by the solution. This column is calculated by checking if the total shortfall associated with each target (i.e. "absolute_shortfall" column) is equal to zero.

Solution format

Broadly speaking, the argument to solution must be in the same format as the planning unit data in the argument to x. Further details on the correct format are listed separately for each of the different planning unit data formats:

x **has** numeric **planning units** The argument to solution must be a numeric vector with each element corresponding to a different planning unit. It should have the same number of planning units as those in the argument to x. Additionally, any planning units missing cost (NA) values should also have missing (NA) values in the argument to solution.

x **has** matrix **planning units** The argument to solution must be a matrix vector with each row corresponding to a different planning unit, and each column correspond to a different management zone. It should have the same number of planning units and zones as those in the argument to x. Additionally, any planning units missing cost (NA) values for a particular zone should also have a missing (NA) values in the argument to solution.

x **has** Raster **planning units** The argument to solution be a Raster object where different grid cells (pixels) correspond to different planning units and layers correspond to a different management zones. It should have the same dimensionality (rows, columns, layers), resolution, extent, and coordinate reference system as the planning units in the argument to x. Additionally, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

x **has** data.frame **planning units** The argument to solution must be a data.frame with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if a data.frame object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with sf::sf() data). Additionally, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

x **has** Spatial **planning units** The argument to solution must be a Spatial object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the Spatial object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example with sf::sf() data). Additionally, the argument to solution must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

x **has** sf::sf() **planning units** The argument to solution must be a sf::sf() object with each column corresponding to a different zone, each row corresponding to a different planning unit, and cell values corresponding to the solution value. This means that if the sf::sf() object containing the solution also contains additional columns, then these columns will need to be subsetted prior to using this function (see below for example). Additionally, the argument to solution must also have the same coordinate reference system as the planning unit data. Furthermore, any planning units missing cost (NA) values for a particular zone should also have missing (NA) values in the argument to solution.

See Also

problem(), summaries.

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
data(sim_pu_raster, sim_pu_sf, sim_features,
     sim_pu_zones_sf, sim_features_zones)

# build minimal conservation problem with raster data
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve the problem
```

```
s1 <- solve(p1)

# print solution
print(s1)

# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)

# calculate target coverage by the solution
r1 <- eval_target_coverage_summary(p1, s1)
print(r1, width = Inf) # note: `width = Inf` tells R to print all columns

# build minimal conservation problem with polygon (sf) data
p2 <- problem(sim_pu_sf, sim_features, cost_column = "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve the problem
s2 <- solve(p2)

# print first six rows of the attribute table
print(head(s2))

# plot solution
plot(s2[, "solution_1"])

# calculate target coverage by the solution
r2 <- eval_target_coverage_summary(p2, s2[, "solution_1"])
print(r2, width = Inf)

# build multi-zone conservation problem with polygon (sf) data
p3 <- problem(sim_pu_zones_sf, sim_features_zones,
              cost_column = c("cost_1", "cost_2", "cost_3")) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve the problem
s3 <- solve(p3)

# print first six rows of the attribute table
print(head(s3))

# create new column representing the zone id that each planning unit
# was allocated to in the solution
s3$solution <- category_vector(
  s3[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")])
s3$solution <- factor(s3$solution)
```

```
# plot solution
plot(s3[, "solution"])

# calculate target coverage by the solution
r3 <- eval_target_coverage_summary(
  p3, s3[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")])
print(r3, width = Inf)

# create a new column with character values containing the zone names,
# by extracting these data out of the zone column
# (which is in list-column format)
r3$zone2 <- vapply(r3$zone, FUN.VALUE = character(1), paste, sep = " & ")

# print r3 again to show the new column
print(r3, width = Inf)

## End(Not run)
```

`fast_extract` *Fast extract*

Description

Extract data from a `Raster` object.

Usage

```
fast_extract(x, y, ...)

## S4 method for signature 'Raster,SpatialPolygons'
fast_extract(x, y, fun = "mean", ...)

## S4 method for signature 'Raster,SpatialPoints'
fast_extract(x, y, fun = "mean", ...)

## S4 method for signature 'Raster,SpatialLines'
fast_extract(x, y, fun = "mean", ...)

## S4 method for signature 'Raster,sfc'
fast_extract(x, y, fun = "mean", ...)

## S4 method for signature 'Raster,sf'
fast_extract(x, y, fun = "mean", ...)
```

Arguments

x	`Raster` object.
y	`Spatial` or `sf::sf()` object.

`...` not used.

`fun` `character` name of statistic to summarize data. Defaults to `"mean"`. Available options include `"sum"` or `"mean"`. Defaults to `"mean"`.

Details

This function is simply a wrapper that uses `raster::extract()` to extract data for `SpatialPoints` and `SpatialLines` and non-polygonal `sf::sf()` data, and `exactextractr::exact_extract()` for `SpatialPolygons` and polygonal `sf::sf()` data.

Value

`matrix` containing the summary amount of each feature within each planning unit. Rows correspond to different spatial features in the argument to `y` and columns correspond to different raster layers in the argument to `x`.

See Also

`raster::extract()`, `exactextractr::exact_extract()`.

Examples

```
# load data
data(sim_pu_sf, sim_features)

# extract data
result <- fast_extract(sim_features, sim_pu_sf)

# show result
print(head(result))
```

`feature_abundances` *Feature abundances*

Description

Calculate the total abundance of each feature found in the planning units of a conservation planning problem.

Usage

```
feature_abundances(x, na.rm)

## S3 method for class 'ConservationProblem'
feature_abundances(x, na.rm = FALSE)
```

Arguments

x	problem() (i.e. ConservationProblem) object.
na.rm	logical should planning units with NA cost data be excluded from the abundance calculations? The default argument is FALSE.

Details

Planning units can have cost data with finite values (e.g. 0.1, 3, 100) and NA values. This functionality is provided so that locations which are not available for protected area acquisition can be included when calculating targets for conservation features (e.g. when targets are specified using add_relative_targets()). If the total amount of each feature in all the planning units is required—including the planning units with NA cost data—then the the na.rm argument should be set to FALSE. However, if the planning units with NA cost data should be excluded—for instance, to calculate the highest feasible targets for each feature—then the na.rm argument should be set to TRUE.

Value

tibble::tibble() object containing the total amount ("absolute_abundance") and proportion ("relative_abundance") of the distribution of each feature in the planning units. Here, each row contains data that pertain to a specific feature in a specific management zone (if multiple zones are present). This object contains the following columns:

feature character name of the feature.

zone character name of the zone (not included when the argument to x contains only one management zone).

absolute_abundance numeric amount of each feature in the planning units. If the problem contains multiple zones, then this column shows how well each feature is represented in a each zone.

relative_abundance numeric proportion of the feature's distribution in the planning units. If the argument to na.rm is FALSE, then this column will only contain values equal to one. Otherwise, if the argument to na.rm is TRUE and planning units with NA cost data contain non-zero amounts of each feature, then this column will contain values between zero and one.

See Also

problem(), eval_feature_representation_summary().

Examples

```
# load data
data(sim_pu_raster, sim_features)

# create a simple conservation planning dataset so we can see exactly
# how the feature abundances are calculated
pu <- data.frame(id = seq_len(10), cost = c(0.2, NA, runif(8)),
                 spp1 = runif(10), spp2 = c(rpois(9, 4), NA))

# create problem
```

```
p1 <- problem(pu, c("spp1", "spp2"), cost_column = "cost")

# calculate feature abundances; including planning units with NA costs
a1 <- feature_abundances(p1, na.rm = FALSE) # (default)
print(a1)

# calculate feature abundances; excluding planning units with NA costs
a2 <- feature_abundances(p1, na.rm = TRUE)
print(a2)

# verify correctness of feature abundance calculations
all.equal(a1$absolute_abundance,
          c(sum(pu$spp1), sum(pu$spp2, na.rm = TRUE)))

all.equal(a1$relative_abundance,
          c(sum(pu$spp1) / sum(pu$spp1),
            sum(pu$spp2, na.rm = TRUE) / sum(pu$spp2, na.rm = TRUE)))

all.equal(a2$absolute_abundance,
          c(sum(pu$spp1[!is.na(pu$cost)]),
            sum(pu$spp2[!is.na(pu$cost)], na.rm = TRUE)))

all.equal(a2$relative_abundance,
          c(sum(pu$spp1[!is.na(pu$cost)]) / sum(pu$spp1, na.rm = TRUE),
            sum(pu$spp2[!is.na(pu$cost)], na.rm = TRUE) / sum(pu$spp2,
                                                              na.rm = TRUE)))

# initialize conservation problem with raster data
p3 <- problem(sim_pu_raster, sim_features)

# calculate feature abundances; including planning units with NA costs
a3 <- feature_abundances(p3, na.rm = FALSE) # (default)
print(a3)

# create problem using total amounts of features in all the planning units
# (including units with NA cost data)
p4 <- p3 %>%
      add_min_set_objective() %>%
      add_relative_targets(a3$relative_abundance) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# attempt to solve the problem, but we will see that this problem is
# infeasible because the targets cannot be met using only the planning units
# with finite cost data
## Not run:
s4 <- try(solve(p4))

## End(Not run)
# calculate feature abundances; excluding planning units with NA costs
a5 <- feature_abundances(p3, na.rm = TRUE)
print(a5)
```

```
# create problem using total amounts of features in the planning units with
# finite cost data
p5 <- p3 %>%
      add_min_set_objective() %>%
      add_relative_targets(a5$relative_abundance) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve the problem
s5 <- solve(p5)

# plot the solution
# this solution contains all the planning units with finite cost data (i.e.
# cost data that do not have NA values)
plot(s5)

## End(Not run)
```

---

feature_names *Feature names*

Description

Extract the names of the features in an object.

Usage

```
feature_names(x)

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

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

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

Arguments

x	`problem()` (i.e. `ConservationProblem`) or `Zones()` object.

Value

`character` feature names.

Examples

```
# load data
data(sim_pu_raster, sim_features)

# create problem
p <- problem(sim_pu_raster, sim_features) %>%
     add_min_set_objective() %>%
     add_relative_targets(0.2) %>%
     add_binary_decisions()

# print feature names
print(feature_names(p))
```

---

importance *Evaluate solution importance*

Description

Importance scores (also known as irreplaceability scores) can be used to assess the relative importance of planning units selected in a solution to a conservation planning `problem()`.

Details

The following methods are available for calculating importance scores:

`eval_replacement_importance()` Calculate importance scores using replacement costs (based on Cabeza and Moilanen 2006). These scores quantify the change in the objective function (e.g. additional costs required to meet feature targets) of the optimal solution if a given planning unit in a solution cannot be acquired. They can (i) account for the cost of different planning units, (ii) account for multiple management zones, (iii) apply to any objective function, and (iv) identify truly irreplaceable planning units (denoted with infinite values).

`eval_ferrier_importance()` Calculate importance scores following Ferrier *et al.* (2000). These scores measure importance based on how critical planning units are for meeting targets. They can only be applied to conservation problems with a minimum set objective and a single zone (i.e. the classic *Marxan*-type problem). Furthermore—unlike the replacement cost scores—these scores provide a score for each feature within each planning unit, providing insight into why certain planning units are more important than other planning units.

`eval_rare_richness_importance()` Calculate importance scores using the rarity weighted richness metric (based on Williams *et al.* 1996). These score are simply a measure of biodiversity. They do not account for planning costs, multiple management zones, objective functions, or feature targets (or weightings). They merely describe the spatial patterns of biodiversity, and do not account for many of the factors needed to quantify the importance of a planning unit for achieving conservation goals.

Generally speaking, we recommend using replacement cost scores for small and moderate sized problems (e.g. less than 30,000 planning units) when it is feasible to do so. It can take a very long time to compute replacement cost scores, and so it is simply not feasible to compute these scores for particularly large problems. For large sized problems (e.g. more than 100,000 planning units), we recommend using the rarity weighted richness scores simply because there is no other choice available. It has been known for decades that such measures of biodiversity lead to poor conservation plans (Kirkpatrick 1983). We do not currently recommend using the Ferrier scores because the code requires further testing to verify correctness.

References

Cabeza M and Moilanen A (2006) Replacement cost: A practical measure of site value for cost-effective reserve planning. *Biological Conservation*, 132: 336–342.

Ferrier S, Pressey RL, and Barrett TW (2000) A new predictor of the irreplaceability of areas for achieving a conservation goal, its application to real-world planning, and a research agenda for further refinement. *Biological Conservation*, 93: 303–325.

Kirkpatrick, JB (1983) An iterative method for establishing priorities for the selection of nature reserves: an example from Tasmania. *Biological Conservation*, 25: 127–134.

Williams P, Gibbons D, Margules C, Rebelo A, Humphries C, and Pressey RL (1996) A comparison of richness hotspots, rarity hotspots and complementary areas for conserving diversity using British birds. *Conservation Biology*, 10: 155–174.

See Also

problem().

Examples

```
## Not run:
# load data
data(sim_pu_raster, sim_pu_polygons, sim_features)

# build minimal conservation problem with raster data
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(gap = 0, verbose = FALSE)

# solve the problem
s1 <- solve(p1)

# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)

# calculate importance scores using replacement cost scores
ir1 <- eval_replacement_importance(p1, s1)

# calculate importance scores using Ferrier et al 2000 method,
# and extract the total importance scores
ir2 <- eval_ferrier_importance(p1, s1)[["total"]]

# calculate importance scores using rarity weighted richness scores
```

```
ir3 <- eval_rare_richness_importance(p1, s1)

# plot importance scores
plot(stack(ir1, ir2, ir3), axes = FALSE, box = FALSE,
     main = c("replacement cost", "Ferrier score",
              "rarity weighted richness"))

## End(Not run)
```

---

intersecting_units *Find intersecting units*

Description

Find which of the units in a spatial data object intersect with the units in another spatial data object.

Usage

```
intersecting_units(x, y)

## S4 method for signature 'Raster,Raster'
intersecting_units(x, y)

## S4 method for signature 'Spatial,Spatial'
intersecting_units(x, y)

## S4 method for signature 'sf,Spatial'
intersecting_units(x, y)

## S4 method for signature 'Spatial,Raster'
intersecting_units(x, y)

## S4 method for signature 'Spatial,sf'
intersecting_units(x, y)

## S4 method for signature 'Raster,Spatial'
intersecting_units(x, y)

## S4 method for signature 'sf,sf'
intersecting_units(x, y)

## S4 method for signature 'Raster,sf'
intersecting_units(x, y)

## S4 method for signature 'sf,Raster'
intersecting_units(x, y)

## S4 method for signature 'data.frame,ANY'
intersecting_units(x, y)
```

Arguments

x	Spatial or Raster object.
y	Spatial or Raster object.

Value

integer indices of the units in x that intersect with y.

See Also

fast_extract().

Examples

```
# create data
r <- raster(matrix(1:9, byrow = TRUE, ncol=3))
r_with_holes <- r
r_with_holes[c(1, 5, 9)] <- NA
ply <- rasterToPolygons(r)
ply_with_holes <- st_as_sf(rasterToPolygons(r_with_holes))

# intersect raster with raster
## Not run:
par(mfrow = c(1, 2))
plot(r, main = "x=Raster")
plot(r_with_holes, main = "y=Raster")

## End(Not run)
print(intersecting_units(r, r_with_holes))

# intersect raster with polygons (sf)
## Not run:
par(mfrow = c(1, 2))
plot(r, main = "x=Raster")
plot(ply_with_holes, main = "y=sf", key.pos = NULL, reset = FALSE)

## End(Not run)
print(intersecting_units(r, ply_with_holes))

# intersect polygons (Spatial) with raster
## Not run:
par(mfrow = c(1, 2))
plot(ply, main = "x=Spatial")
plot(r_with_holes, main = "y=Raster")

## End(Not run)
print(intersecting_units(ply, r_with_holes))

# intersect polygons (Spatial) with polygons (sf)
## Not run:
par(mfrow = c(1, 2))
```

```
plot(ply, main = "x=Spatial")
plot(ply_with_holes, main = "y=sf", key.pos = NULL, reset = FALSE)

## End(Not run)
print(intersecting_units(ply, ply_with_holes))
```

`is.Id` *Is it?*

Description

Test if an object inherits from a class.

Usage

```
is.Id(x)

is.Waiver(x)
```

Arguments

`x`	Object.

Value

`logical` indicating if it inherits from the class.

`loglinear_interpolation` *Log-linear interpolation*

Description

Log-linearly interpolate values between two thresholds.

Usage

```
loglinear_interpolation(
  x,
  coordinate_one_x,
  coordinate_one_y,
  coordinate_two_x,
  coordinate_two_y
)
```

Arguments

x — numeric *x* values for which interpolate *y* values.

coordinate_one_x — numeric value for lower *x*-coordinate.

coordinate_one_y — numeric value for lower *y*-coordinate.

coordinate_two_x — numeric value for upper *x*-coordinate.

coordinate_two_y — numeric value for upper *y*-coordinate.

Details

Values are log-linearly interpolated at the *x*-coordinates specified in x using the lower and upper coordinate arguments to define the line. Values lesser or greater than these numbers are assigned the minimum and maximum *y* coordinates.

Value

numeric values.

Examples

```
# create series of x-values
x <- seq(0, 1000)

# interpolate y-values for the x-values given the two reference points:
# (200, 100) and (900, 15)
y <- loglinear_interpolation(x, 200, 100, 900, 15)

# plot the interpolated values
## Not run:
plot(y ~ x)

# add the reference points to the plot (shown in red)
points(x = c(200, 900), y = c(100, 15), pch = 18, col = "red", cex = 2)

## End(Not run)

# this function can also be used to calculate representation targets
# following Rodrigues et al. (2014). For example, let's say that
# we had a set of species we were interested in calculating representation
# targets for and we had information on their range sizes (in km^2).
spp_range_size_km2 <- seq(0.01, 15000000, by = 100)

# we can now use this function to calculate representation targets
# (expressed as a percentage of the species' range sizes) using
# the thresholds and cap sizes reported by Rodrigues et al. 2014
spp_target_percentage_rodrigues <-
  loglinear_interpolation(
```

```
    x = spp_range_size_km2,
    coordinate_one_x = 1000,
    coordinate_one_y = 1,
    coordinate_two_x = 250000,
    coordinate_two_y = 0.1) * 100

# it is also common to apply a cap to the representation targets,
# so let's apply the cap these targets following Butchart et al. (2015)
spp_target_percentage_butchart <- ifelse(
  spp_range_size_km2 >= 10000000,
  (1000000 / spp_range_size_km2) * 100,
  spp_target_percentage_rodrigues)

# plot species range sizes and representation targets
plot(spp_target_percentage_butchart ~ spp_range_size_km2,
  xlab = "Range size km^2" , ylab = "Representation target (%)", type = "l")

# plot species range sizes and representation targets on a log10 scale
plot(spp_target_percentage_butchart ~ log10(spp_range_size_km2),
  xlab = "Range size km^2" , ylab = "Representation target (%)",
  type = "l", xaxt = "n")
axis(1, pretty(log10(spp_range_size_km2)),
     10^pretty(log10(spp_range_size_km2)))
```

---

marxan_boundary_data_to_matrix

Convert Marxan *boundary data to a matrix format*

Description

Convert a `data.frame` object that follows the *Marxan* format to a matrix format. This function is useful for converting `data.frame` objects to `matrix` or `array` objects that are used by the various penalties and constraints functions. If the boundary data contains data for a single zone, then a matrix object is returned. Otherwise if the boundary data contains data for multiple zones, then an array is returned.

Usage

```
marxan_boundary_data_to_matrix(x, data)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object that contains planning unit and zone data to ensure that the argument to `data` is converted correctly. This argument can be set to `NULL` if checks are not required (not recommended).
data	`data.frame` object with the columns `"id1"`, `"id2"`, and `"boundary"`. The columns `"zone1"` and `"zone2"` can also be provided to indicate zone data.

Value

array or dgCMatrix sparse matrix object.

Examples

```
# create marxan boundary with four planning units and one zone
bldf1 <- expand.grid(id1 = seq_len(4), id2 = seq_len(4))
bldf1$boundary <- 1
bldf1$boundary[bldf1$id1 == bldf1$id2] <- 0.5

# convert to matrix
m1 <- marxan_boundary_data_to_matrix(NULL, bldf1)

# visualize matrix
## Not run:
image(m1)

## End(Not run)
# create marxan boundary with three planning units and two zones
bldf2 <- expand.grid(id1 = seq_len(3), id2 = seq_len(3),
                     zone1 = c("z1", "z2"),
                     zone2 = c("z1", "z2"))
bldf2$boundary <- 1
bldf2$boundary[bldf2$id1 == bldf2$id2 & bldf2$zone1 == bldf2$zone2] <- 0.5
bldf2$boundary[bldf2$id1 == bldf2$id2 & bldf2$zone1 != bldf2$zone2] <- 0

# convert to array
m2 <- marxan_boundary_data_to_matrix(NULL, bldf2)

# print array
print(m2)
```

---

marxan_problem   Marxan *conservation problem*

Description

Create a conservation planning problem() following the mathematical formulations used in *Marxan* (detailed in Beyer *et al.* 2016). Note that these problems are solved using exact algorithms and not simulated annealing (i.e. the *Marxan* software).

Usage

```
marxan_problem(x, ...)

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

## S3 method for class 'data.frame'
```

```
marxan_problem(x, spec, puvspr, bound = NULL, blm = 0, ...)

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

Arguments

x
: `character` file path for a *Marxan* input file (typically called `"input.dat"`), or `data.frame` containing planning unit data (typically called `"pu.dat"`). If the argument to `x` is a `data.frame`, then each row corresponds to a different planning unit, and it must have the following columns:

  id `integer` unique identifier for each planning unit. These identifiers are used in the argument to `puvspr`.

  cost `numeric` cost of each planning unit.

  status `integer` indicating if each planning unit should not be locked in the solution (0) or if it should be locked in (2) or locked out (3) of the solution. Although *Marxan* allows planning units to be selected in the initial solution (using values of 1), these values have no effect here. This column is optional.

...
: not used.

spec
: `data.frame` containing information on the features. The argument to `spec` must follow the conventions used by *Marxan* for the species data file (conventionally called `"spec.dat"`). Each row corresponds to a different feature and each column corresponds to different information about the features. It must contain the columns listed below. Note that the argument to `spec` must contain at least one column named `"prop"` or `"amount"`—**but not both columns with both of these names**—to specify the target for each feature.

  id `integer` unique identifier for each feature These identifiers are used in the argument to `puvspr`.

  name `character` name for each feature.

  prop `numeric` relative target for each feature (optional).'

  amount `numeric` absolute target for each feature (optional).

puvspr
: `data.frame` containing information on the amount of each feature in each planning unit. The argument to `puvspr` must follow the conventions used in the *Marxan* input data file (conventionally called `"puvspr.dat"`). It must contain the following columns:

  pu `integer` planning unit identifier.

  species `integer` feature identifier.

  amount `numeric` amount of the feature in the planning unit.

bound
: `NULL` object indicating that no boundary data is required for the conservation planning problem, or a `data.frame` containing information on the planning units' boundaries. The argument to `bound` must follow the conventions used in the *Marxan* input data file (conventionally called `"bound.dat"`). It must contain the following columns:

  id1 `integer` planning unit identifier.

id2 `integer` planning unit identifier.

boundary `numeric` length of shared boundary between the planning units identified in the previous two columns.

blm — `numeric` boundary length modifier. This argument only has an effect when argument to `x` is a `data.frame`. The default argument is zero.

Details

This function is provided as a convenient wrapper for solving *Marxan* problems using **prioritizr**.

Value

`problem()` (i.e. `ConservationProblem`) object.

Notes

In early versions, this function could accommodate asymmetric connectivity data. This functionality is no longer supported. To specify asymmetric connectivity, please see the `add_connectivity_penalties()` function.

References

Ball IR, Possingham HP, and Watts M (2009) *Marxan and relatives: Software for spatial conservation prioritisation* in Spatial conservation prioritisation: Quantitative methods and computational tools. Eds Moilanen A, Wilson KA, and Possingham HP. Oxford University Press, Oxford, UK.

Beyer HL, Dujardin Y, Watts ME, and Possingham HP (2016) Solving conservation planning problems with integer linear programming. *Ecological Modelling*, 228: 14–22.

See Also

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

Examples

```
# create Marxan problem using Marxan input file
input_file <- system.file("extdata/input.dat", package = "prioritizr")
p1 <- marxan_problem(input_file) %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s1 <- solve(p1)

# print solution
head(s1)

## End(Not run)
# create Marxan problem using data.frames that have been loaded into R
## load in planning unit data
pu_path <- system.file("extdata/input/pu.dat", package = "prioritizr")
```

```
pu_dat <- data.table::fread(pu_path, data.table = FALSE)
head(pu_dat)

## load in feature data
spec_path <- system.file("extdata/input/spec.dat", package = "prioritizr")
spec_dat <- data.table::fread(spec_path, data.table = FALSE)
head(spec_dat)

## load in planning unit vs feature data
puvspr_path <- system.file("extdata/input/puvspr.dat",
                           package = "prioritizr")
puvspr_dat <- data.table::fread(puvspr_path, data.table = FALSE)
head(puvspr_dat)

## load in the boundary data
bound_path <- system.file("extdata/input/bound.dat", package = "prioritizr")
bound_dat <- data.table::fread(bound_path, data.table = FALSE)
head(bound_dat)

# create problem without the boundary data
p2 <- marxan_problem(pu_dat, spec_dat, puvspr_dat) %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s2 <- solve(p2)

# print solution
head(s2)

## End(Not run)
# create problem with the boundary data and a boundary length modifier
# set to 5
p3 <- marxan_problem(pu_dat, spec_dat, puvspr_dat, bound_dat, 5) %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s3 <- solve(p3)

# print solution
head(s3)

## End(Not run)
```

---

matrix_parameters *Matrix parameters*

Description

Create a parameter that represents a matrix object.

Usage

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

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

Arguments

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

Value

`MiscParameter` object.

Examples

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

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

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

# create a binary matrix parameter
p2 <- binary_matrix_parameter("m", m)
print(p2) # print it
```

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

MiscParameter-class   *Miscellaneous parameter prototype*

Description

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

Fields

$id `character` identifier for parameter.

$name `character` name of parameter.

$value `tibble::tibble()` object.

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

Usage

`x$print()`

`x$show()`

`x$validate(x)`

`x$get()`

`x$set(x)`

`x$reset()`

Arguments

x object used to set a new parameter value.

Details

print print the object.

show show the object.

validate check if a proposed new parameter is valid.

get extract the parameter value.

set update the parameter value.

reset update the parameter value to be the default value.

See Also

`Parameter`.

`misc_parameter` *Miscellaneous parameter*

Description

Create a parameter that consists of a miscellaneous object.

Usage

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

Arguments

`name`	`character` name of parameter.
`value`	object.
`validator`	`function` to validate changes to the parameter. This function must have a single argument and return either `TRUE` or `FALSE` depending on if the argument is valid candidate for the parameter.

Value

`MiscParameter` object.

Examples

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

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

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

```
iris3 <- iris; iris2[1,1] <- 100 # create updated iris dataset
p2$set(iris3) # set parameter to iris3
p2$print() # print it again
```

---

new_id   *Identifier*

Description

Generate a new unique identifier.

Usage

```
new_id()
```

Details

Identifiers are made using the `uuid::UUIDgenerate()`.

Value

`Id` object.

See Also

`uuid::UUIDgenerate()`.

Examples

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

# print id
print(i)

# convert to character
as.character(i)

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

new_optimization_problem *Optimization problem*

Description

Generate a new empty OptimizationProblem object.

Usage

```
new_optimization_problem()
```

Value

OptimizationProblem object.

See Also

OptimizationProblem-methods

Examples

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

# print new object
print(x)
```

new_waiver *Waiver*

Description

Create a waiver object.

Usage

```
new_waiver()
```

Details

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

Value

Object of class `Waiver`.

Examples

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

# print object
print(w)

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

`number_of_features` *Number of features*

Description

Extract the number of features in an object.

Usage

```
number_of_features(x)

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

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

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

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

Arguments

x	`problem()` (i.e. `ConservationProblem`), `OptimizationProblem`, or `Zones()` object.

Value

`integer` number of features.

Examples

```
# load data
data(sim_pu_raster, sim_features)

# create problem
p <- problem(sim_pu_raster, sim_features) %>%
     add_min_set_objective() %>%
     add_relative_targets(0.2) %>%
     add_binary_decisions()

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

---

number_of_planning_units
Number of planning units

Description

Extract the number of planning units in an object.

Usage

```
number_of_planning_units(x)

## S4 method for signature 'ConservationProblem'
number_of_planning_units(x)

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

Arguments

x	`problem()` (i.e. `ConservationProblem`), `OptimizationProblem`, or `Zones()` object.

Value

`integer` number of planning units.

Examples

```
# load data
data(sim_pu_raster, sim_features)

# create problem
p <- problem(sim_pu_raster, sim_features) %>%
     add_min_set_objective() %>%
```

```
        add_relative_targets(0.2) %>%
        add_binary_decisions()

# print number of planning units
print(number_of_planning_units(p))
```

`number_of_total_units` *Number of total units*

Description

Extract the number of total units in an object.

Usage

```
number_of_total_units(x)

## S4 method for signature 'ConservationProblem'
number_of_total_units(x)
```

Arguments

x   problem() (i.e. ConservationProblem), OptimizationProblem, or Zones() object.

Value

integer number of total units.

Examples

```
# load data
data(sim_pu_raster, sim_pu_zones_stack, sim_features, sim_features_zones)

# create problem with one zone
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_binary_decisions()

# print number of planning units
print(number_of_planning_units(p1))

# print number of total units
print(number_of_total_units(p1))

# create problem with multiple zones
p2 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
```

```
      add_relative_targets(matrix(0.2, ncol = 3, nrow = 5)) %>%
      add_binary_decisions()

# print number of planning units
print(number_of_planning_units(p2))

# print number of total units
print(number_of_total_units(p2))
```

---

number_of_zones *Number of zones*

Description

Extract the number of zones in an object.

Usage

```
number_of_zones(x)

## S4 method for signature 'ConservationProblem'
number_of_zones(x)

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

## S4 method for signature 'ZonesRaster'
number_of_zones(x)

## S4 method for signature 'ZonesCharacter'
number_of_zones(x)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`), `OptimizationProblem`, or `Zones()` object.

Value

`integer` number of zones.

Examples

```
# load data
data(sim_pu_zones_stack, sim_features_zones)

# print number of zones in a Zones object
print(number_of_zones(sim_features_zones))
# create problem with multiple zones
```

```
p <- problem(sim_pu_zones_stack, sim_features_zones) %>%
     add_min_set_objective() %>%
     add_relative_targets(matrix(0.2, ncol = 3, nrow = 5)) %>%
     add_binary_decisions()

# print number of zones in the problem
print(number_of_zones(p))
```

Objective-class *Objective prototype*

Description

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

---

objectives *Add an objective*

Description

An objective is used to specify the overall goal of a conservation planning problem(). All conservation planning problems involve minimizing or maximizing some kind of objective. For instance, the planner may require a solution that conserves enough habitat for each species while minimizing the overall cost of the reserve network. Alternatively, the planner may require a solution that maximizes the number of conserved species while ensuring that the cost of the reserve network does not exceed the budget.

Details

Please note that failing to specify an objective before attempting to solve a problem will return an error.

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

add_min_set_objective() Minimize the cost of the solution whilst ensuring that all targets are met. This objective is similar to that used in *Marxan*.

add_max_cover_objective() Represent at least one instance of as many features as possible within a given budget.

add_max_features_objective() Fulfill as many targets as possible while ensuring that the cost of the solution does not exceed a budget.

add_min_shortfall_objective() Minimize the overall (weighted sum) shortfall for as many targets as possible while ensuring that the cost of the solution does not exceed a budget.

add_min_largest_shortfall_objective() Minimize the largest (maximum) shortfall among all targets while ensuring that the cost of the solution does not exceed a budget.

add_max_phylo_div_objective() Maximize the phylogenetic diversity of the features represented in the solution subject to a budget.

add_max_phylo_end_objective() Maximize the phylogenetic endemism of the features represented in the solution subject to a budget.

add_max_utility_objective() Secure as much of the features as possible without exceeding a budget.

See Also

constraints, decisions, penalties, portfolios, problem(), solvers, targets.

Examples

```
# load data
data(sim_pu_raster, sim_features, sim_phylogeny)

# create base problem
p <- problem(sim_pu_raster, sim_features) %>%
     add_relative_targets(0.1) %>%
     add_binary_decisions() %>%
     add_default_solver(verbose = FALSE)

 # create problem with added minimum set objective
p1 <- p %>% add_min_set_objective()

# create problem with added maximum coverage objective
# note that this objective does not use targets
p2 <- p %>% add_max_cover_objective(500)

# create problem with added maximum feature representation objective
p3 <- p %>% add_max_features_objective(1900)

# create problem with added minimum shortfall objective
p4 <- p %>% add_min_shortfall_objective(1900)

# create problem with added minimum largest shortfall objective
p5 <- p %>% add_min_largest_shortfall_objective(1900)

# create problem with added maximum phylogenetic diversity objective
p6 <- p %>% add_max_phylo_div_objective(1900, sim_phylogeny)

# create problem with added maximum phylogenetic diversity objective
p7 <- p %>% add_max_phylo_end_objective(1900, sim_phylogeny)

# create problem with added maximum utility objective
# note that this objective does not use targets
p8 <- p %>% add_max_utility_objective(1900)

## Not run:
# solve problems
s <- stack(solve(p1), solve(p2), solve(p3), solve(p4), solve(p5), solve(p6),
           solve(p7), solve(p8))
```

```
# plot solutions
plot(s, axes = FALSE, box = FALSE,
     main = c("min set", "max coverage", "max features",
              "min shortfall", "min largest shortfall",
              "max phylogenetic diversity",
              "max phylogenetic endemism", "max utility"))

## End(Not run)
```

OptimizationProblem-class *Optimization problem class*

Description

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

Fields

$ptr `externalptr` object.

Usage

```
x$print()
x$show()
x$repr()
x$ncol()
x$nrow()
x$ncell()
x$modelsense()
x$vtype()
x$obj()
x$A()
x$rhs()
x$sense()
x$lb()
x$ub()
x$number_of_planning_units()
x$number_of_features()
x$number_of_zones()
```

```
x$row_ids()

x$col_ids()

x$compressed_formulation()
```

Arguments

ptr `externalptr` object.

Details

print print the object.

show show the object.

repr `character` representation of object.

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

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

ncell `integer` number of cells in model matrix.

modelsense `character` model sense.

vtype `character` vector of variable types.

obj `numeric` vector of objective function.

A `dgCMatrix` model matrix

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

sense `character` vector of constraint senses.

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

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

number_of_features `integer` number of features in the problem.

number_of_planning_units `integer` number of planning units in the problem.

number_of_zones `integer` number of zones in the problem.

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

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

compressed_formulation is the optimization problem formulated using a compressed version of the rij matrix?

shuffle_columns randomly shuffle the columns in the problem. This should almost never be called manually and only should only be called after the optimization problem has been fully constructed.

OptimizationProblem-methods

Optimization problem methods

Description

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

Usage

```
nrow(x)

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

ncol(x)

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

ncell(x)

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

modelsense(x)

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

vtype(x)

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

obj(x)

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

A(x)

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

rhs(x)
```

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

sense(x)

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

lb(x)

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

ub(x)

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

col_ids(x)

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

row_ids(x)

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

compressed_formulation(x)

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

Arguments

x	`OptimizationProblem` object.

Details

The functions return the following data:

nrow `integer` number of rows (constraints).

ncol `integer` number of columns (decision variables).

ncell `integer` number of cells.

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

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

obj `numeric` vector specifying the objective function.

A `dgCMatrix` matrix object defining the problem matrix.

rhs `numeric` vector with right-hand-side linear constraints

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

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

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

number_of_planning_units `integer` number of planning units in the problem.

number_of_features `integer` number of features the problem.

Value

`dgCMatrix`, `numeric` vector, `numeric` vector, or scalar `integer` depending on the method used.

`Parameter-class` *Parameter class*

Description

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

Fields

$id `Id` identifier for parameter.

$name `character` name of parameter.

$value `numeric vector` of values.

$default `numeric vector` of default values.

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

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

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

Usage

`x$print()`

`x$show()`

`x$reset()`

Details

print print the object.

show show the object.

reset change the parameter values to be the default values.

See Also

ScalarParameter.

---

parameters *Parameters*

Description

Create a new collection of Parameter objects.

Usage

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

Arguments

...	Parameter objects.

Value

Parameters object.

See Also

array_parameters(), scalar_parameters().

Examples

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

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

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

Parameters-class *Parameters class*

Description

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

Fields

$parameters `list` object containing `Parameter` objects.

Usage

`x$print()`

`x$show()`

`x$repr()`

`x$names()`

`x$ids()`

`x$length()`

`x$get(id)`

`x$set(id,value)`

`x$add(p)`

Arguments

id `Id` object.

p `Parameter` object.

value any object.

Details

print print the object.

show show the object.

repr `character` representation of object.

names return `character` names of parameters.

ids return `character` parameter unique identifiers.

length return `integer` number of parameters in object.

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

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

penalties *Add a penalty*

Description

A penalty can be applied to a conservation planning `problem()` to penalize solutions according to a specific metric. Penalties—unlike constraints—act as an explicit trade-off with the objective being minimized or maximized (e.g. solution cost when used with `add_min_set_objective()`).

Details

Both penalties and constraints can be used to modify a problem and identify solutions that exhibit specific characteristics. Constraints work by invalidating solutions that do not exhibit specific characteristics. On the other hand, penalties work by specifying trade-offs against the main problem objective and are mediated by a penalty factor.

The following penalties can be added to a conservation planning `problem()`:

`add_boundary_penalties()` Add penalties to a conservation problem to favor solutions that have planning units clumped together into contiguous areas.

`add_connectivity_penalties()` Add penalties to a conservation problem to favor solutions that select planning units with high connectivity between them.

`add_linear_penalties()` Add penalties to a conservation problem to favor solutions that avoid selecting planning units based on a certain variable (e.g. anthropogenic pressure).

See Also

constraints, decisions, objectives portfolios, `problem()`, solvers, targets.

Examples

```
# load data
data(sim_pu_raster, sim_features)

# create basic problem
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_default_solver(verbose = FALSE)

# create problem with boundary penalties
p2 <- p1 %>% add_boundary_penalties(5, 1)

# create connectivity matrix based on spatial proximity
 scm <- as.data.frame(sim_pu_raster, xy = TRUE, na.rm = FALSE)
 scm <- 1 / (as.matrix(dist(scm)) + 1)

# remove weak and moderate connections between planning units to reduce
# run time
```

```
scm[scm < 0.85] <- 0

# create problem with connectivity penalties
p3 <- p1 %>% add_connectivity_penalties(25, data = scm)

# create problem with linear penalties,
# here the penalties will be based on random numbers to keep it simple

# simulate penalty data
sim_penalty_raster <- simulate_cost(sim_pu_raster)

# plot penalty data
plot(sim_penalty_raster, main = "penalty data", axes = FALSE, box = FALSE)

# create problem with linear penalties, with a penalty scaling factor of 100
p4 <- p1 %>% add_linear_penalties(100, data = sim_penalty_raster)

## Not run:
# solve problems
s <- stack(solve(p1), solve(p2), solve(p3), solve(p4))

# plot solutions
plot(s, axes = FALSE, box = FALSE,
     main = c("basic solution", "boundary penalties",
              "connectivity penalties", "linear penalties"))

## End(Not run)
```

`Penalty-class` *Penalty prototype*

Description

This prototype is used to represent penalties that are added to the objective function when making a conservation problem. **This prototype represents a recipe, to actually add penalties to a planning problem, see the help page on penalties. Only experts should use this class directly.** This prototype inherits from the `ConservationModifier`.

See Also

`ConservationModifier`.

`Portfolio-class` *Portfolio prototype*

Description

This prototype is used to represent methods for generating portfolios of optimization problems. **This class represents a recipe to create portfolio generating method and is only recommended for use by expert users. To customize the method used to generate portfolios, please see the help page on portfolios.**

Fields

$name `character` name of portfolio method.

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

$run `function` used to generate a portfolio.

Usage

`x$print()`

`x$show()`

`x$repr()`

`x$run(op,sol)`

Arguments

x `Solver` object.

op `OptimizationProblem` object.

Details

print print the object.

show show the object.

repr `character` representation of object.

run solve an `OptimizationProblem` object using this object and a `Solver` object.

portfolios *Solution portfolios*

Description

Conservation planning exercises rarely have access to all the data needed to identify the *truly* perfect solution. This is because available data may lack important details (e.g. land acquisition costs may be unavailable), contain errors (e.g. species presence/absence data may have false positives), or key objectives may not be formally incorporated into the prioritization process (e.g. future land use requirements). As such, conservation planners can help decision makers by providing them with a portfolio of solutions to inform their decision.

Details

The following portfolios can be added to a conservation planning `problem()`. Note that all methods for generating portfolios return solutions that are within the specified optimality gap.

`add_extra_portfolio()` Generate a portfolio of solutions by storing feasible solutions found during the optimization process. This method is useful for quickly obtaining multiple solutions, but does not provide any guarantees on the number of solutions, or the quality of solutions. Note that it requires the *Gurobi* solver.

`add_top_portfolio()` Generate a portfolio of solutions by finding a pre-specified number of solutions that are closest to optimality (i.e the top solutions). This is useful for examining differences among near-optimal solutions. It can also be used to generate multiple solutions and, in turn, to calculate selection frequencies for small problems. Note that it requires the *Gurobi* solver.

`add_gap_portfolio()` Generate a portfolio of solutions by finding a certain number of solutions that are all within a pre- specified optimality gap. This method is useful for generating multiple solutions that can be used to calculate selection frequencies for moderate and large-sized problems (similar to *Marxan*). Note that it requires the *Gurobi* solver.

`add_cuts_portfolio()` Generate a portfolio of distinct solutions within a pre-specified optimality gap using Bender's cuts. This is recommended as a replacement for `add_top_portfolio()` when the *Gurobi* software is not available.

`add_shuffle_portfolio()` Generate a portfolio of solutions by randomly reordering the data prior to attempting to solve the problem. This is recommended as a replacement for `add_gap_portfolio()` when the *Gurobi* software is not available.

See Also

constraints, decisions, objectives penalties, `problem()`, solvers, targets.

Examples

```
# load data
data(sim_pu_raster, sim_features)
```

```
# create problem
p <- problem(sim_pu_raster, sim_features) %>%
     add_min_set_objective() %>%
     add_relative_targets(0.1) %>%
     add_binary_decisions() %>%
     add_default_solver(gap = 0.02, verbose = FALSE)

# create problem with cuts portfolio with 4 solutions
p1 <- p %>% add_cuts_portfolio(4)

# create problem with shuffle portfolio with 4 solutions
p2 <- p %>% add_shuffle_portfolio(4)
## Not run:
# create problem with extra portfolio
p3 <- p %>% add_extra_portfolio()

# create problem with top portfolio with 4 solutions
p4 <- p %>% add_top_portfolio(4)

# create problem with gap portfolio with 4 solutions within 50% of optimality
p5 <- p %>% add_gap_portfolio(4, 0.5)

# solve problems and create solution portfolios
s <- list(solve(p1), solve(p2), solve(p3), solve(p4), solve(p5))

# plot solutions from extra portfolio
plot(stack(s[[1]]), axes = FALSE, box = FALSE)

# plot solutions from top portfolio
plot(stack(s[[2]]), axes = FALSE, box = FALSE)

# plot solutions from gap portfolio
plot(stack(s[[3]]), axes = FALSE, box = FALSE)

# plot solutions from cuts portfolio
plot(stack(s[[4]]), axes = FALSE, box = FALSE)

# plot solutions from shuffle portfolio
plot(stack(s[[5]]), axes = FALSE, box = FALSE)


## End(Not run)
```

pproto *Create a new* `pproto` *object*

Description

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

Usage

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

Arguments

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

Examples

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

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

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

---

predefined_optimization_problem

Predefined optimization problem

Description

Create a new `OptimizationProblem` object.

Usage

```
predefined_optimization_problem(x)
```

Arguments

`x`	`list` object containing data to construct the problem.

Details

The argument to x must be a list that contains the following elements:

modelsense character model sense.

number_of_features integer number of features in problem.

number_of_planning_units integer number of planning units.

A_i integer row indices for problem matrix.

A_j integer column indices for problem matrix.

A_x numeric values for problem matrix.

obj numeric objective function values.

lb numeric lower bound for decision values.

ub numeric upper bound for decision values.

rhs numeric right-hand side values.

sense numeric constraint senses.

vtype character variable types. These are used to specify that the decision variables are binary ("B") or continuous ("C").

row_ids character identifiers for the rows in the problem matrix.

col_ids character identifiers for the columns in the problem matrix.

Examples

```
# create list with problem data
l <- list(modelsense = "min", number_of_features = 2,
          number_of_planning_units = 3, number_of_zones = 1,
          A_i = c(0L, 1L, 0L, 1L, 0L, 1L), A_j = c(0L, 0L, 1L, 1L, 2L, 2L),
          A_x = c(2, 10, 1, 10, 1, 10), obj = c(1, 2, 2), lb = c(0, 1, 0),
          ub = c(0, 1, 1), rhs = c(2, 10), compressed_formulation = TRUE,
          sense = c(">=", ">="), vtype = c("B", "B", "B"),
          row_ids = c("spp_target", "spp_target"),
          col_ids = c("pu", "pu", "pu"))

# create OptimizationProblem object
x <- predefined_optimization_problem(l)

# print new object
print(x)
```

presolve_check *Presolve check*

Description

Check a conservation planning problem() for potential issues before trying to solve it. Specifically, problems are checked for (i) values that are likely to result in "strange" solutions and (ii) values that are likely to cause numerical instability issues and lead to unreasonably long run times when solving it. Although these checks are provided to help diagnose potential issues, please be aware that some detected issues may be false positives. Please note that these checks will not be able to verify if a problem has a feasible solution or not.

Usage

```
presolve_check(x)

## S3 method for class 'ConservationProblem'
presolve_check(x)

## S3 method for class 'OptimizationProblem'
presolve_check(x)
```

Arguments

x	problem() (i.e. ConservationProblem) or OptimizationProblem object.

Details

This function checks for issues that are likely to result in "strange" solutions. Specifically, it checks if (i) all planning units are locked in, (ii) all planning units are locked out, and (iii) all planning units have negative cost values (after applying penalties if any were specified). Although such conservation planning problems are mathematically valid, they are generally the result of a coding mistake when building the problem (e.g. using an absurdly high penalty value or using the wrong dataset to lock in planning units). Thus such issues, if they are indeed issues and not false positives, can be fixed by carefully checking the code, data, and parameters used to build the conservation planning problem.

This function then checks for values that may lead to numerical instability issues when solving the problem. Specifically, it checks if the range of values in certain components of the optimization problem are over a certain threshold (i.e. 1×10^9) or if the values themselves exceed a certain threshold (i.e. 1×10^{10}). In most cases, such issues will simply cause an exact algorithm solver to take a very long time to generate a solution. In rare cases, such issues can cause incorrect calculations which can lead to exact algorithm solvers returning infeasible solutions (e.g. a solution to the minimum set problem where not all targets are met) or solutions that exceed the specified optimality gap (e.g. a suboptimal solution when a zero optimality gap is specified).

What can you do if a conservation planning problem fails to pass these checks? Well, this function will have thrown some warning messages describing the source of these issues, so read them carefully. For instance, a common issue is when a relatively large penalty value is specified for boundary (add_boundary_penalties()) or connectivity penalties (add_connectivity_penalties()). This can be fixed by trying a smaller penalty value. In such cases, the original penalty value supplied was so high that the optimal solution would just have selected every single planning unit in the solution—and this may not be especially helpful anyway (see below for example). Another common issue is that the planning unit cost values are too large. For example, if you express the costs of the planning units in terms of USD then you might have some planning units that cost over one billion dollars in large-scale planning exercises. This can be fixed by rescaling the values so that they are smaller (e.g. multiplying the values by a number smaller than one, or expressing them as a fraction of the maximum cost). Let's consider another common issue, let's pretend that you used habitat suitability models to predict the amount of suitable habitat in each planning unit for each feature. If you calculated the amount of suitable habitat in each planning unit in square meters then this could lead to very large numbers. You could fix this by converting the units from square meters to square kilometers or thousands of square kilometers. Alternatively, you could calculate the percentage of each planning unit that is occupied by suitable habitat, which will yield values between zero and one hundred.

But what can you do if you can't fix these issues by simply changing the penalty values or rescaling data? You will need to apply some creative thinking. Let's run through a couple of scenarios. Let's pretend that you have a few planning units that cost a billion times more than any other planning unit so you can't fix this by rescaling the cost values. In this case, it's extremely unlikely that these planning units will be selected in the optimal solution so just set the costs to zero and lock them out. If this procedure yields a problem with no feasible solution, because one (or several) of the planning units that you manually locked out contains critical habitat for a feature, then find out which planning unit(s) is causing this infeasibility and set its cost to zero. After solving the problem, you will need to manually recalculate the cost of the solutions but at least now you can be confident that you have the optimal solution. Now let's pretend that you are using the maximum features objective (i.e. add_max_features_objective()) and assigned some really high weights to the targets for some features to ensure that their targets were met in the optimal solution. If you set the weights for these features to one billion then you will probably run into numerical instability issues. Instead, you can calculate minimum weight needed to guarantee that these features will be represented in the optimal solution and use this value instead of one billion. This minimum weight value can be calculated as the sum of the weight values for the other features and adding a small number to it (e.g. 1). Finally, if you're running out of ideas for addressing numerical stability issues you have one remaining option: you can use the `numeric_focus` argument in the add_gurobi_solver() function to tell the solver to pay extra attention to numerical instability issues. This is not a free lunch, however, because telling the solver to pay extra attention to numerical issues can substantially increase run time. So, if you have problems that are already taking an unreasonable time to solve, then this will not help at all.

Value

`logical` value indicating if all checks are passed successfully.

See Also

problem(), solve(), https://www.gurobi.com/documentation/8.1/refman/numerics_gurobi_guidelines.html, http://files.gurobi.com/Numerics.pdf.

Examples

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

# load data
data(sim_pu_raster, sim_features)

# create minimal problem with no issues
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions()

# run presolve checks
# note that no warning is thrown which suggests that we should not
# encounter any numerical stability issues when trying to solve the problem
print(presolve_check(p1))

# create a minimal problem, containing cost values that are really
# high so that they could cause numerical instability issues when trying
# to solve it
sim_pu_raster2 <- sim_pu_raster
sim_pu_raster2[1] <- 1e+15
p2 <- problem(sim_pu_raster2, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions()

# run presolve checks
# note that a warning is thrown which suggests that we might encounter
# some issues, such as long solve time or suboptimal solutions, when
# trying to solve the problem
print(presolve_check(p2))

# create a minimal problem with connectivity penalties values that have
# a really high penalty value that is likely to cause numerical instability
# issues when trying to solve the it
cm <- adjacency_matrix(sim_pu_raster)
p3 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_connectivity_penalties(1e+15, data = cm) %>%
      add_binary_decisions()

# run presolve checks
# note that a warning is thrown which suggests that we might encounter
# some numerical instability issues when trying to solve the problem
print(presolve_check(p3))
## Not run:
# let's forcibly solve the problem using Gurobi and tell it to
# be extra careful about numerical instability problems
s3 <- p3 %>%
```

```
      add_gurobi_solver(numeric_focus = TRUE) %>%
      solve(force = TRUE)

# plot solution
# we can see that all planning units were selected because the connectivity
# penalty is so high that cost becomes irrelevant, so we should try using
# a much lower penalty value
plot(s3, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
```

print *Print*

Description

Display information about an object.

Usage

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

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

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

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

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

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

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

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

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

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

Arguments

x	Any object.
...	not used.

Value

None.

See Also

`base::print()`.

Examples

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

`prioritizr` *prioritizr: Systematic Conservation Prioritization in R*

Description

The **prioritizr R** package uses mixed integer linear programming (MILP) techniques to provide a flexible interface for building and solving conservation planning problems (Rodrigues *et al.* 2000; Billionnet 2013). It supports a broad range of objectives, constraints, and penalties that can be used to custom-tailor conservation planning problems to the specific needs of a conservation planning exercise. Once built, conservation planning problems can be solved using a variety of commercial and open-source exact algorithm solvers. In contrast to the algorithms conventionally used to solve conservation problems, such as heuristics or simulated annealing (Ball *et al.* 2009), the exact algorithms used here are guaranteed to find optimal solutions. Furthermore, conservation problems can be constructed to optimize the spatial allocation of different management actions or zones, meaning that conservation practitioners can identify solutions that benefit multiple stakeholders. Finally, this package has the functionality to read input data formatted for the *Marxan* conservation planning program (Ball *et al.* 2009), and find much cheaper solutions in a much shorter period of time than *Marxan* (Beyer *et al.* 2016). See the online code repository for more information.

Details

This package contains several vignettes that are designed to showcase its functionality. To view them, please use the code `vignette("name",package = "prioritizr")` where `"name"` is the name of the desired vignette (e.g. `"gurobi_installation"`).

prioritizr provides background information on systematic conservation planning and a comprehensive overview of the package and its usage.

gurobi_installation contains detailed instructions for installing and setting up the *Gurobi* software suite for use with the package.

publication_record lists of scientific publications that have used the package for developing prioritizations.

zones describes how problems can be constructed with multiple management actions or zones.

tasmania provides a tutorial using Tasmania, Australia as a case-study. This tutorial uses vector-based planning unit data and is written for individuals familiar with the *Marxan* decision support tool.

saltspring provides a tutorial using Salt Spring Island, Canada as a case-study. This tutorial uses raster-based planning unit data.

References

Ball IR, Possingham HP, and Watts M (2009) *Marxan and relatives: Software for spatial conservation prioritisation* in Spatial conservation prioritisation: Quantitative methods and computational tools. Eds Moilanen A, Wilson KA, and Possingham HP. Oxford University Press, Oxford, UK.

Beyer HL, Dujardin Y, Watts ME, and Possingham HP (2016) Solving conservation planning problems with integer linear programming. *Ecological Modelling*, 228: 14–22.

Billionnet A (2013) Mathematical optimization ideas for biodiversity conservation. *European Journal of Operational Research*, 231: 514–534.

Rodrigues AS, Cerdeira OJ, and Gaston KJ (2000) Flexibility, efficiency, and accountability: adapting reserve selection algorithms to more complex conservation problems. *Ecography*, 23: 565–574.

`prioritizr-deprecated` *Deprecation notice*

Description

The functions listed here are deprecated. This means that they once existed in earlier versions of the of the **prioritizr** 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 **prioritizr** 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

```
add_connected_constraints(...)

add_corridor_constraints(...)

set_number_of_threads(...)

get_number_of_threads(...)

is.parallel(...)
```

```
add_pool_portfolio(...)

connected_matrix(...)

feature_representation(...)

replacement_cost(...)

rarity_weighted_richness(...)

ferrier_score(...)
```

Arguments

`...` not used.

Details

The following functions have been deprecated:

`add_connected_constraints()` renamed as the `add_contiguity_constraints()` function.

`add_corridor_constraints()` replaced by the `add_feature_contiguity_constraints()` function.

`set_number_of_threads()` no longer needed used with the implementation of superior data extraction.

`get_number_of_threads()` no longer needed used with the implementation of superior data extraction.

`is.parallel()` no longer needed used with the implementation of superior data extraction

`add_pool_portfolio()` replaced by the `add_extra_portfolio()` and `add_top_portfolio()`.

`connected_matrix()` renamed as the `adjacency_matrix()` function.

`feature_representation()` replaced by the `eval_feature_representation_summary()` function for consistency with other functions.

`replacement_cost()` renamed as the `eval_replacement_importance()` function for consistency with other functions for evaluating solutions.

`rarity_weighted_richness()` renamed as the `eval_rare_richness_importance()` function for consistency with other functions for evaluating solutions.

`ferrier_score()` renamed as the `eval_ferrier_importance()` function for consistency with other functions for evaluating solutions.

problem *Conservation planning problem*

Description

Create a systematic conservation planning problem. This function is used to specify the basic data used in a spatial prioritization problem: the spatial distribution of the planning units and their costs, as well as the features (e.g. species, ecosystems) that need to be conserved. After constructing this `ConservationProblem-class` object, it can be customized to meet specific goals using objectives, targets, constraints, and penalties. After building the problem, the `solve()` function can be used to identify solutions.

Usage

```
problem(x, features, ...)

## S4 method for signature 'Raster,Raster'
problem(x, features, run_checks, ...)

## S4 method for signature 'Raster,ZonesRaster'
problem(x, features, run_checks, ...)

## S4 method for signature 'Spatial,Raster'
problem(x, features, cost_column, run_checks, ...)

## S4 method for signature 'Spatial,ZonesRaster'
problem(x, features, cost_column, run_checks, ...)

## S4 method for signature 'Spatial,character'
problem(x, features, cost_column, ...)

## S4 method for signature 'Spatial,ZonesCharacter'
problem(x, features, cost_column, ...)

## S4 method for signature 'data.frame,character'
problem(x, features, cost_column, ...)

## S4 method for signature 'data.frame,ZonesCharacter'
problem(x, features, cost_column, ...)

## S4 method for signature 'data.frame,data.frame'
problem(x, features, rij, cost_column, zones, ...)

## S4 method for signature 'numeric,data.frame'
problem(x, features, rij_matrix, ...)

## S4 method for signature 'matrix,data.frame'
```

```
problem(x, features, rij_matrix, ...)

## S4 method for signature 'sf,Raster'
problem(x, features, cost_column, run_checks, ...)

## S4 method for signature 'sf,ZonesRaster'
problem(x, features, cost_column, run_checks, ...)

## S4 method for signature 'sf,character'
problem(x, features, cost_column, ...)

## S4 method for signature 'sf,ZonesCharacter'
problem(x, features, cost_column, ...)
```

Arguments

`x` Raster, sf::st_sf(), SpatialPolygonsDataFrame, SpatialLinesDataFrame, SpatialPointsDataFrame, data.frame() object, numeric() vector, or matrix() specifying the planning units to use in the reserve design exercise and their corresponding cost. It may be desirable to exclude some planning units from the analysis, for example those outside the study area. To exclude planning units, set the cost for those raster cells to NA, or use the add_locked_out_constraint function.

`features` The feature data can be specified in a variety of ways. The specific formats that can be used depend on the cost data format (i.e. argument to x) and whether the problem should have a single zone or multiple zones. If the problem should have a single zone, then the feature data can be specified following:

- x = RasterLayer-class, or x = Spatial-class, or x = sf::st_sf(): y = Raster-class object showing the distribution of conservation features. Missing values (i.e. NA values) can be used to indicate the absence of a feature in a particular cell instead of explicitly setting these cells to zero. Note that this argument type for features can only be used to specify data for problems involving a single zone.
- x = Spatial-class, or x = sf::st_sf(), or x = data.frame: y = character vector with column names that correspond to the abundance or occurrence of different features in each planning unit. Note that this argument type can only be used to create problems involving a single zone.
- x = Spatial-class, or x = sf::st_sf(), or x = data.frame, or x = numeric vector, or x = matrix: y = data.frame object containing the names of the features. Note that if this type of argument is supplied to features then the argument rij or rij_matrix must also be supplied. This type of argument should follow the conventions used by *Marxan*, wherein each row corresponds to a different feature. It must also contain the following columns:

  id integer unique identifier for each feature These identifiers are used in the argument to rij.

  name character name for each feature.

  prop numeric relative target for each feature (optional).

amount numeric absolute target for each feature (optional).

If the problem should have multiple zones, then the feature data can be specified following:

- x = RasterStack-class, or x = RasterBrick-class, or x = Spatial-class, or x = sf::st_sf(): y = ZonesRaster: object showing the distribution of conservation features in multiple zones. As above, missing values (i.e. NA values) can be used to indicate the absence of a feature in a particular cell instead of explicitly setting these cells to zero.
- x = Spatial-class, or x = sf::st_sf(), or or x = data.frame: y = ZonesCharacter object with column names that correspond to the abundance or occurrence of different features in each planning unit in different zones.

... not used.

run_checks logical flag indicating whether checks should be run to ensure the integrity of the input data. These checks are run by default; however, for large datasets they may increase run time. If it is taking a prohibitively long time to create the prioritization problem, it is suggested to try setting run_checks to FALSE.

cost_column character name or integer indicating the column(s) with the cost data. This argument must be supplied when the argument to x is a Spatial or data.frame object. This argument should contain the name of each column containing cost data for each management zone when creating problems with multiple zones. To create a problem with a single zone, then set the argument to cost_column as a single column name.

rij data.frame containing information on the amount of each feature in each planning unit assuming each management zone. Similar to data.frame arguments for features, the data.frame objects must follow the conventions used by *Marxan*. Note that the "zone" column is not needed for problems involving a single management zone. Specifically, the argument should contain the following columns:

pu integer planning unit identifier.

species integer feature identifier.

zone integer zone identifier (optional for problems involving a single zone).

amount numeric amount of the feature in the planning unit.

zones data.frame containing information on the zones. This argument is only used when argument to x and y are both data.frame objects and the problem being built contains multiple zones. Following conventions used in MarZone, this argument should contain the following columns: columns:

id integer zone identifier.

name character zone name.

rij_matrix list of matrix or dgCMatrix objects specifying the amount of each feature (rows) within each planning unit (columns) for each zone. The list elements denote different zones, matrix rows denote features, and matrix columns denote planning units. For convenience, the argument to rij_matrix can be a single matrix or dgCMatrix when specifying a problem with a single management zone. This argument is only used when the argument to x is a numeric or matrix object.

Details

A reserve design exercise starts by dividing the study region into planning units (typically square or hexagonal cells) and, for each planning unit, assigning values that quantify socioeconomic cost and conservation benefit for a set of conservation features. The cost can be the acquisition cost of the land, the cost of management, the opportunity cost of foregone commercial activities (e.g. from logging or agriculture), or simply the area. The conservation features are typically species (e.g. Clouded Leopard) or habitats (e.g. mangroves or cloud forest). The benefit that each feature derives from a planning unit can take a variety of forms, but is typically either occupancy (i.e. presence or absence) or area of occurrence within each planning unit. Finally, in some types of reserve design models, representation targets must be set for each conservation feature, such as 20% of the current extent of cloud forest or 10,000 km^2 of Clouded Leopard habitat (see targets).

The goal of the reserve design exercise is then to optimize the trade-off between conservation benefit and socioeconomic cost, i.e. to get the most benefit for your limited conservation funds. In general, the goal of an optimization problem is to minimize an objective function over a set of decision variables, subject to a series of constraints. The decision variables are what we control, usually there is one binary variable for each planning unit specifying whether or not to protect that unit (but other approaches are available, see decisions). The constraints can be thought of as rules that need to be followed, for example, that the reserve must stay within a certain budget or meet the representation targets.

Integer linear programming (ILP) is the subset of optimization algorithms used in this package to solve reserve design problems. The general form of an integer programming problem can be expressed in matrix notation using the following equation.

$$Minimize \mathbf{c}^{\mathbf{T}}\mathbf{x} subject to \mathbf{A}\mathbf{x} \geq= or \leq \mathbf{b}$$

Here, x is a vector of decision variables, c and b are vectors of known coefficients, and A is the constraint matrix. The final term specifies a series of structural constraints where relational operators for the constraint can be either $\geq$, $=$, or $\leq$ the coefficients. For example, in the minimum set cover problem, c would be a vector of costs for each planning unit, b a vector of targets for each conservation feature, the relational operator would be $\geq$ for all features, and A would be the representation matrix with $A_{ij} = r_{ij}$, the representation level of feature i in planning unit j.

Please note that this function internally computes the amount of each feature in each planning unit when this data is not supplied (using the `rij_matrix` parameter). As a consequence, it can take a while to initialize large-scale conservation planning problems that involve millions of planning units.

Value

`ConservationProblem` object containing data for building a prioritization problem.

See Also

constraints, decisions, objectives penalties, portfolios, solvers, targets, summaries, importance.

Examples

```
# load data
```

```
data(sim_pu_raster, sim_pu_polygons, sim_pu_lines, sim_pu_points,
     sim_pu_sf, sim_features)

# create problem using raster planning unit data
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

## Not run:
# create problem using polygon (Spatial) planning unit data
p2 <- problem(sim_pu_polygons, sim_features, "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create problem using line (Spatial) planning unit data
p3 <- problem(sim_pu_lines, sim_features, "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create problem using point (Spatial) planning unit data
p4 <- problem(sim_pu_points, sim_features, "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# create problem using polygon (sf) planning unit data
p5 <- problem(sim_pu_sf, sim_features, "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# add columns to polygon planning unit data representing the abundance
# of species inside them
sim_pu_polygons$spp_1 <- rpois(length(sim_pu_polygons), 5)
sim_pu_polygons$spp_2 <- rpois(length(sim_pu_polygons), 8)
sim_pu_polygons$spp_3 <- rpois(length(sim_pu_polygons), 2)

# create problem using pre-processed data when feature abundances are
# stored in the columns of an attribute table for a spatial vector dataset
p6 <- problem(sim_pu_polygons, features = c("spp_1", "spp_2", "spp_3"),
              "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
```

```
# alternatively one can supply pre-processed aspatial data
costs <- sim_pu_polygons$cost
features <- data.frame(id = seq_len(nlayers(sim_features)),
                       name = names(sim_features))
rij_mat <- rij_matrix(sim_pu_polygons, sim_features)
p7 <- problem(costs, features, rij_matrix = rij_mat) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.2) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve problems
s1 <- solve(p1)
s2 <- solve(p2)
s3 <- solve(p3)
s4 <- solve(p4)
s5 <- solve(p5)
s6 <- solve(p6)
s7 <- solve(p7)

# plot solutions for problems associated with spatial data
par(mfrow = c(3, 2), mar = c(0, 0, 4.1, 0))
plot(s1, main = "raster data", axes = FALSE, box = FALSE)

plot(s2, main = "polygon data")
plot(s2[s2$solution_1 == 1, ], col = "darkgreen", add = TRUE)

plot(s3, main = "line data")
lines(s3[s3$solution_1 == 1, ], col = "darkgreen", lwd = 2)

plot(s4, main = "point data", pch = 19)
points(s4[s4$solution_1 == 1, ], col = "darkgreen", cex = 2, pch = 19)

plot(s5, main = "sf (polygon) data", pch = 19)
points(s5[s5$solution_1 == 1, ], col = "darkgreen", cex = 2, pch = 19)

plot(s6, main = "preprocessed data", pch = 19)
plot(s6[s6$solution_1 == 1, ], col = "darkgreen", add = TRUE)

# show solutions for problems associated with aspatial data
str(s7)

## End(Not run)
# create some problems with multiple zones

# first, create a matrix containing the targets for multi-zone problems
# here each row corresponds to a different feature, each
# column corresponds to a different zone, and values correspond
# to the total (absolute) amount of a given feature that needs to be secured
# in a given zone
targets <- matrix(rpois(15, 1),
                  nrow = number_of_features(sim_features_zones),
```

```
                  ncol = number_of_zones(sim_features_zones),
                  dimnames = list(feature_names(sim_features_zones),
                                  zone_names(sim_features_zones)))

# print targets
print(targets)

# create a multi-zone problem with raster data
p8 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_absolute_targets(targets) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve problem
s8 <- solve(p8)

# plot solution
# here, each layer/panel corresponds to a different zone and pixel values
# indicate if a given planning unit has been allocated to a given zone
par(mfrow = c(1, 1))
plot(s8, main = c("zone 1", "zone 2", "zone 3"), axes = FALSE, box = FALSE)

# alternatively, the category_layer function can be used to create
# a new raster object containing the zone ids for each planning unit
# in the solution (note this only works for problems with binary decisions)
par(mfrow = c(1, 1))
plot(category_layer(s8), axes = FALSE, box = FALSE)

# create a multi-zone problem with polygon data
p9 <- problem(sim_pu_zones_polygons, sim_features_zones,
              cost_column = c("cost_1", "cost_2", "cost_3")) %>%
      add_min_set_objective() %>%
      add_absolute_targets(targets) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve problem
s9 <- solve(p9)

# create column containing the zone id for which each planning unit was
# allocated to in the solution
s9$solution <- category_vector(s9@data[, c("solution_1_zone_1",
                                           "solution_1_zone_2",
                                           "solution_1_zone_3")])
s9$solution <- factor(s9$solution)

# plot solution
spplot(s9, zcol = "solution", main = "solution", axes = FALSE, box = FALSE)

# create a multi-zone problem with polygon planning unit data
# and where fields (columns) in the attribute table correspond
# to feature abundances
```

```
# first fields need to be added to the planning unit data
# which indicate the amount of each feature in each zone
# to do this, the fields will be populated with random counts
sim_pu_zones_polygons$spp1_z1 <- rpois(nrow(sim_pu_zones_polygons), 1)
sim_pu_zones_polygons$spp2_z1 <- rpois(nrow(sim_pu_zones_polygons), 1)
sim_pu_zones_polygons$spp3_z1 <- rpois(nrow(sim_pu_zones_polygons), 1)
sim_pu_zones_polygons$spp1_z2 <- rpois(nrow(sim_pu_zones_polygons), 1)
sim_pu_zones_polygons$spp2_z2 <- rpois(nrow(sim_pu_zones_polygons), 1)
sim_pu_zones_polygons$spp3_z2 <- rpois(nrow(sim_pu_zones_polygons), 1)

# create problem with polygon planning unit data and use field names
# to indicate feature data
# additionally, to make this example slightly more interesting,
# the problem will have prfoportion-type decisions such that
# a proportion of each planning unit can be allocated to each of the
# two management zones
p10 <- problem(sim_pu_zones_polygons,
               zones(c("spp1_z1", "spp2_z1", "spp3_z1"),
                     c("spp1_z2", "spp2_z2", "spp3_z2"),
                     zone_names = c("z1", "z2")),
               cost_column = c("cost_1", "cost_2")) %>%
       add_min_set_objective() %>%
       add_absolute_targets(targets[1:3, 1:2]) %>%
       add_proportion_decisions() %>%
      add_default_solver(verbose = FALSE)

# solve problem
s10 <- solve(p10)

# plot solution
spplot(s10, zcol = c("solution_1_z1", "solution_1_z2"), main = "solution",
       axes = FALSE, box = FALSE)

## End(Not run)
```

`proximity_matrix` *Proximity matrix*

Description

Create a matrix showing which planning units are within a certain spatial proximity to each other.

Usage

```
proximity_matrix(x, distance)

## S3 method for class 'Raster'
proximity_matrix(x, distance)
```

```
## S3 method for class 'SpatialPolygons'
proximity_matrix(x, distance)

## S3 method for class 'SpatialLines'
proximity_matrix(x, distance)

## S3 method for class 'SpatialPoints'
proximity_matrix(x, distance)

## S3 method for class 'sf'
proximity_matrix(x, distance)

## Default S3 method:
proximity_matrix(x, distance)
```

Arguments

x	Raster, Spatial, or sf::sf() object representing planning units.
distance	numeric distance threshold. Planning units that are further apart from each other than this threshold are not treated as being within proximity of each other.

Details

Proximity calculations are performed using sf::st_is_within_distance().

Value

dsCMatrix symmetric sparse matrix object. Each row and column represents a planning unit. Cells values indicate if the pair-wise distances between different planning units are within the distance threshold or not (using ones and zeros). To reduce computational burden, cells among the matrix diagonal are set to zero. Furthermore, if the argument to x is a Raster object, then cells with NA values are set to zero too.

Examples

```
# load data
data(sim_pu_raster, sim_pu_sf, sim_pu_lines, sim_pu_points)

# create proximity matrix using raster data
## crop raster to 9 cells to provide a small example
r <- crop(sim_pu_raster, c(0, 0.3, 0, 0.3))

## make proximity matrix using a distance threshold of 2
cm_raster <- proximity_matrix(r, distance = 2)

# create proximity matrix using polygon (sf) data
## subset 9 polygons to provide a small example
ply <- sim_pu_sf[c(1:2, 10:12, 20:22), ]

## make proximity matrix using a distance threshold of 2
```

```
cm_ply <- proximity_matrix(ply, distance = 2)

# create proximity matrix using line (Spatial) data
## subset 9 lines to provide a small example
lns <- sim_pu_lines[c(1:2, 10:12, 20:22), ]

## make proximity matrix
cm_lns <- proximity_matrix(lns, distance = 2)

## create proximity matrix using point (Spatial) data
## subset 9 points to provide a small example
pts <- sim_pu_points[c(1:2, 10:12, 20:22), ]

# make proximity matrix
cm_pts <- proximity_matrix(pts, distance = 2)

# plot data and the proximity matrices
## Not run:
par(mfrow = c(4,2))

## plot raster and proximity matrix
plot(r, main = "raster", axes = FALSE, box = FALSE)
plot(raster(as.matrix(cm_raster)), main = "proximity matrix", axes = FALSE,
     box = FALSE)

## plot polygons and proximity matrix
plot(r, main = "polygons (sf)", axes = FALSE, box = FALSE)
plot(raster(as.matrix(cm_ply)), main = "proximity matrix", axes = FALSE,
    box = FALSE)

## plot lines and proximity matrix
plot(r, main = "lines (Spatial)", axes = FALSE, box = FALSE)
plot(raster(as.matrix(cm_lns)), main = "proximity matrix", axes = FALSE,
     box = FALSE)

## plot points and proximity matrix
plot(r, main = "points (Spatial)", axes = FALSE, box = FALSE)
plot(raster(as.matrix(cm_pts)), main = "proximity matrix", axes = FALSE,
     box = FALSE)

## End(Not run)
```

`rij_matrix` *Feature by planning unit matrix*

Description

Generate a matrix showing the amount of each feature in each planning unit (also known as an *rij* matrix).

Usage

```
rij_matrix(x, y, ...)

## S4 method for signature 'Raster,Raster'
rij_matrix(x, y, ...)

## S4 method for signature 'Spatial,Raster'
rij_matrix(x, y, fun, ...)

## S4 method for signature 'sf,Raster'
rij_matrix(x, y, fun, ...)
```

Arguments

x	`Raster`, `Spatial`, or `sf::sf()` object representing the planning units.
y	`Raster` object representing the features.
...	not used.
fun	`character` for summarizing values inside each planning unit. This parameter is only used when the argument to `x` is a `Spatial` or `sf::sf()` object. Defaults to `"sum"`.

Details

Generally, processing vector (i.e. `Spatial` or `sf::sf()`) data takes much longer to process then `Raster` data, so it is recommended to use `Raster` data for planning units where possible.

Value

`dgCMatrix` sparse matrix object. The sparse matrix represents the spatial intersection between the planning units and the features. Rows correspond to features, and columns correspond to planning units. Values correspond to the amount (or presence/absence) of the feature in the planning unit. For example, the amount of the third species in the second planning unit would be stored in the third column and second row.

Examples

```
# load data
data(sim_pu_raster, sim_pu_polygons, sim_pu_sf, sim_pu_zones_stack)

# create rij matrix using raster layer planning units
rij_raster <- rij_matrix(sim_pu_raster, sim_features)
print(rij_raster)

# create rij matrix using polygon (Spatial) planning units
rij_polygons <- rij_matrix(sim_pu_polygons, sim_features)
print(rij_polygons)

# create rij matrix using polygon (sf) planning units
rij_sf <- rij_matrix(sim_pu_sf, sim_features)
```

```
print(rij_sf)

# create rij matrix using raster stack planning units
rij_zones_raster <- rij_matrix(sim_pu_zones_stack, sim_features)
print(rij_zones_raster)
```

---

run_calculations *Run calculations*

Description

Execute preliminary calculations in a conservation problem and store the results for later use. This function is useful when creating slightly different versions of the same conservation planning problem that involve the same pre-processing steps (e.g. calculating boundary data), because means that the same calculations will not be run multiple times.

Usage

```
run_calculations(x)
```

Arguments

x	`problem()` (i.e. `ConservationProblem`) object.

Details

This function is used for the effect of modifying the input `ConservationProblem` object. As such, it does not return anything. To use this function with `pipe()` operators, use the %T>% operator and not the %>% operator.

Value

Invisible `TRUE` indicating success.

Examples

```
## Not run:
# Let us imagine a scenario where we wanted to understand the effect of
# setting different targets on our solution.

# create a conservation problem with no targets
p <- problem(sim_pu_raster, sim_features) %>%
     add_min_set_objective() %>%
     add_boundary_penalties(10, 0.5) %>%
     add_binary_decisions() %>%
     add_default_solver(verbose = FALSE)

# create a copies of p and add targets
p1 <- p %>% add_relative_targets(0.1)
```

```
p2 <- p %>% add_relative_targets(0.2)
p3 <- p %>% add_relative_targets(0.3)

# now solve each of the different problems and record the time spent
# solving them
s1 <- system.time({solve(p1); solve(p2); solve(p3)})

# This approach is inefficient. Since these problems all share the same
# planning units it is actually performing the same calculations three times.
# To avoid this, we can use the "run_calculations" function before creating
# the copies. Normally, R runs the calculations just before solving the
# problem

# recreate a conservation problem with no targets and tell R run the
# preliminary calculations. Note how we use the %T>% operator here.
p <- problem(sim_pu_raster, sim_features) %>%
     add_min_set_objective() %>%
     add_boundary_penalties(10, 0.5) %>%
     add_binary_decisions() %>%
     add_default_solver(verbose = FALSE) %T>%
     run_calculations()

# create a copies of p and add targets just like before
p1 <- p %>% add_relative_targets(0.1)
p2 <- p %>% add_relative_targets(0.2)
p3 <- p %>% add_relative_targets(0.3)

# solve each of the different problems and record the time spent
# solving them
s2 <- system.time({solve(p1); solve(p2); solve(p3)})

# now lets compare the times
print(s1) # time spent without running preliminary calculations
print(s2) # time spent after running preliminary calculations

# As we can see, we can save time by running the preliminary
# calculations before making copies of the problem with slightly
# different constraints. Although the time saved in this example
# is rather small, this is because the example data are very small.
# We would expect larger time savings for larger datasets.

## End(Not run)
```

ScalarParameter-class *Scalar parameter prototype*

Description

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

Fields

$id `character` identifier for parameter.

$name `character` name of parameter.

$value `numeric` scalar value.

$default `numeric` scalar default value.

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

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

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

Usage

```
x$print()

x$show()

x$validate(x)

x$get()

x$set(x)

x$reset()
```

Arguments

x object used to set a new parameter value.

Details

print print the object.

show show the object.

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

get extract the parameter value.

set update the parameter value.

reset update the parameter value to be the default value.

See Also

`Parameter`, `ArrayParameter`.

scalar_parameters *Scalar parameters*

Description

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

Usage

```
proportion_parameter(name, value)

binary_parameter(name, value)

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

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

Arguments

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

Details

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

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

integer_parameter A parameter that is a `integer`.

numeric_parameter A parameter that is a `double`.

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

Value

ScalarParameter object.

Examples

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

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

# integer parameter
p3 <- integer_parameter('int', 5L) # create new object
print(p3) # print it
p3$get() # get value
p3$id # get id
p3$validate(5.6) # check if 5.6 is a validate input
p3$validate(2L) # check if 2L is a validate input
p3$set(2L) # change value to 2L
print(p3) # print it again

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

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

```
p5$set(90) # change value to 8.9
print(p5) # print it again
```

---

show *Show*

Description

Display information about an object.

Usage

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

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

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

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

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

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

Arguments

x	Any object.

Value

None.

See Also

`methods::show()`.

simulate_cost *Simulate cost data*

Description

This function generates cost layers using random field models. By default, it returns spatially autocorrelated integer values.

Usage

```
simulate_cost(
  x,
  n = 1,
 model = RandomFields::RPpoisson(RandomFields::RMtruncsupport(radius = raster::xres(x)
    * 10, RandomFields::RMgauss())),
  transform = identity,
  ...
)
```

Arguments

x	RasterLayer object to use as a template.
n	integer number of species to simulate.
model	RandomFields::RP() model object to use for simulating data.
transform	function to transform values output from the random fields simulation.
...	additional arguments passed to RandomFields::RFsimulate().

Value

RasterStack object.

See Also

simulate_data().

Examples

```
## Not run:
# create raster
r <- raster(ncol=10, nrow=10, xmn=0, xmx=1, ymn=0, ymx=1)
values(r) <- 1

# simulate data
cost <- simulate_cost(r)

# plot simulated species
plot(cost, main = "simulated cost data")
```

```
## End(Not run)
```

`simulate_data` *Simulate data*

Description

Simulate spatially auto-correlated data.

Usage

```
simulate_data(x, n, model, transform = identity, ...)
```

Arguments

x	`RasterLayer` object to use as a template.
n	`integer` number of species to simulate.
model	`RandomFields::RP()` model object to use for simulating data.
transform	`function` to transform values output from the random fields simulation.
...	additional arguments passed to `RandomFields::RFsimulate()`.

Value

`RasterStack` object with a layer for each species.

See Also

`RandomFields::RFsimulate()`, `simulate_cost()`, `simulate_species()`.

Examples

```
## Not run:
# create raster
r <- raster(ncol=10, nrow=10, xmn=0, xmx=1, ymn=0, ymx=1)
values(r) <- 1

# simulate data using a Gaussian field
d <- simulate_data(r, n = 1, model = RandomFields::RMgauss())

# plot simulated data
plot(d, main = "random Gaussian field")

## End(Not run)
```

simulate_species — *Simulate species habitat suitability data*

Description

Generates a random set of species using random field models. By default, the output will contain values between zero and one.

Usage

```
simulate_species(
  x,
  n = 1,
  model = RandomFields::RMgauss(),
  transform = stats::plogis,
  ...
)
```

Arguments

`x`	`RasterLayer` object to use as a template.
`n`	`integer` number of species to simulate.
`model`	`RandomFields::RP()` model object to use for simulating data.
`transform`	`function` to transform values output from the random fields simulation.
`...`	additional arguments passed to `RandomFields::RFsimulate()`.

Value

`RasterStack` object.

See Also

`simulate_data()`.

Examples

```
## Not run:
# create raster
r <- raster(ncol=10, nrow=10, xmn=0, xmx=1, ymn=0, ymx=1)
values(r) <- 1

# simulate 4 species
spp <- simulate_species(r, 4)

# plot simulated species
plot(spp, main = "simulated species distributions")
```

```
## End(Not run)
```

sim_data *Simulated conservation planning data*

Description

Simulated data for making spatial prioritizations.

Usage

```
data(sim_pu_polygons)

data(sim_pu_zones_polygons)

data(sim_pu_points)

data(sim_pu_lines)

data(sim_pu_sf)

data(sim_pu_zones_sf)

data(sim_pu_raster)

data(sim_locked_in_raster)

data(sim_locked_out_raster)

data(sim_pu_zones_stack)

data(sim_features)

data(sim_features_zones)

data(sim_phylogeny)
```

Format

sim_pu_polygons `SpatialPolygonsDataFrame` object.

sim_pu_zones_polygons `SpatialPolygonsDataFrame` object.

sim_pu_sf `sf::sf()` object.

sim_pu_zones_sf `sf::sf()` object.

sim_pu_lines `SpatialLinesDataFrame` object.

sim_pu_points `SpatialPointsDataFrame` object.

sim_pu_raster RasterLayer object.

sim_pu_zones_stack RasterStack object.

sim_locked_in_raster RasterLayer object.

sim_locked_out_raster RasterLayer object.

sim_features RasterStack object.

sim_features_zones ZonesRaster() object.

sim_phylogeny ape::phylo() object.

Details

sim_pu_raster Planning units are represented as raster data. Pixel values indicate planning unit cost and NA values indicate that a pixel is not a planning unit.

sim_pu_zones_stack Planning units are represented as raster stack data. Each layer indicates the cost for a different management zone. Pixels with NA values in a given zone indicate that a planning unit cannot be allocated to that zone in a solution. Additionally, pixels with NA values in all layers are not a planning unit.

sim_locked_in_raster Planning units are represented as raster data. Pixel values are binary and indicate if planning units should be locked in to the solution.

sim_locked_out_raster Planning units are represented as raster data. Pixel values are binary and indicate if planning units should be locked out from the solution.

sim_pu_polygons Planning units represented as polygon data. The attribute table contains fields (columns) indicating the expenditure required for prioritizing each planning unit ("cost" field), if the planning units should be selected in the solution ("locked_in" field), and if the planning units should never be selected in the solution ("locked_out" field).

sim_pu_points Planning units represented as point data. The attribute table follows the same conventions as for sim_pu_polygons.

sim_pu_lines Planning units represented as line data. The attribute table follows the same conventions as for sim_pu_polygons.

sim_pu_sf Planning units represented as polygon data using the sf::sf() package. The attribute table follows the same conventions as for sim_pu_polygons.

sim_pu_zones_polygons Planning units represented as polygon data. The attribute table contains fields (columns) indicating the expenditure required for prioritizing each planning unit under different management zones ("cost_1", "cost_2", and "cost_3" fields), and a series of fields indicating the value that each planning unit that should be assigned in the solution ("locked_1", "locked_2", "locked_3" fields). In these locked fields, planning units that should not be locked to a specific value are assigned a NA value.

sim_pu_zones_sf Planning units represented as polygon data using the sf::sf() package. The attribute tables follows the same conventions as for sim_pu_zone_polygons.

sim_features The simulated distribution of ten species. Pixel values indicate habitat suitability.

sim_features_zones The simulated distribution for five species under three different management zones.

sim_phylogeny The phylogenetic tree for the ten species.

Examples

```
# load data
data(sim_pu_polygons, sim_pu_lines, sim_pu_points, sim_pu_raster,
     sim_locked_in_raster, sim_locked_out_raster, sim_phylogeny,
     sim_features, sim_pu_sf)

# plot example Spatial-class planning unit data
## Not run:
par(mfrow = c(2, 3))
plot(sim_pu_raster, main = "planning units (raster)")
plot(sim_locked_in_raster, main = "locked in units (raster)")
plot(sim_locked_out_raster, main = "locked out units (raster)")
plot(sim_pu_polygons, main = "planning units (polygons)")
plot(sim_pu_lines, main = "planning units (lines)")
plot(sim_pu_points, main = "planning units (points)")

# plot example sf-class planning unit data
plot(sim_pu_sf)

# plot example phylogeny data
par(mfrow = c(1, 1))
ape::plot.phylo(sim_phylogeny, main = "simulated phylogeny")

# plot example feature data
par(mfrow = c(1, 1))
plot(sim_features)

# plot example management zone cost data
par(mfrow = c(1, 1))
plot(sim_pu_zones_stack)

# plot example feature data for each management zone
plot(do.call(stack, sim_features_zones),
     main = paste0("Species ",
                   rep(seq_len(number_of_zones(sim_features_zones)),
                       number_of_features(sim_features_zones)),
                   " (zone ",
                   rep(seq_len(number_of_features(sim_features_zones)),
                       each = number_of_zones(sim_features_zones)),
                   ")"))

## End(Not run)
```

---

solve *Solve*

Description

Solve a conservation planning problem().

Usage

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

## S4 method for signature 'ConservationProblem,missing'
solve(a, b, ..., run_checks = TRUE, force = FALSE)
```

Arguments

a	`problem()` (i.e. `ConservationProblem`) or `OptimizationProblem` object.
b	`Solver` object. Not used if a is an `ConservationProblem` object.
...	arguments passed to `compile()`.
run_checks	`logical` flag indicating whether presolve checks should be run prior solving the problem. These checks are performed using the `presolve_check()` function. Defaults to `TRUE`. Skipping these checks may reduce run time for large problems.
force	`logical` flag indicating if an attempt to should be made to solve the problem even if potential issues were detected during the presolve checks. Defaults to `FALSE`.

Details

After formulating a conservation planning `problem()`, it can be solved using an exact algorithm solver (see solvers for available solvers). If no solver has been explicitly specified, then the best available exact algorithm solver will be used by default (see `add_default_solver()`. Although these exact algorithm solvers will often display a lot of information that isn't really that helpful (e.g. nodes, cutting planes), they do display information about the progress they are making on solving the problem (e.g. the performance of the best solution found at a given point in time). If potential issues were detected during the presolve checks (see `presolve_check()`) and the problem is being forcibly solved (i.e. with `force = TRUE`), then it is also worth checking for any warnings displayed by the solver to see if these potential issues are actually causing issues (e.g. *Gurobi* can display warnings that include `"Warning: Model contains large matrix coefficient range"` and `"Warning: Model contains large rhs"`).

The 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 `logical vector` showing the status of each planning unit in each zone. However, in most cases, the argument to a is an `ConservationProblem` object, and so the type of object returned depends on the number of solutions generated and the type data used to represent the planning units:

`numeric vector` containing the solution. Here, Each element corresponds to a different planning unit. If multiple solutions are generated, then the solution is returned as a `list` of `numeric vectors`.

`matrix` containing `numeric` values for the solution. Here, rows correspond to different planning units, and fields (columns) correspond to different management zones. If multiple solutions are generated, then the solution is returned as a `list` of `matrix` objects.

`Raster` object containing the solution in pixel values. If the argument to x contains a single management zone, then a `RasterLayer` object will be returned. Otherwise, if the argument to x contains multiple zones, then a RasterStack object will be returned containing a different layer for each management zone. If multiple solutions are generated, then the solution is returned as a list of Raster objects.

Spatial, sf::sf(), or data.frame containing the solution in fields (columns). Here, each row corresponds to a different planning unit. If the argument to x contains a single zone, the fields containing solutions are named "solution_XXX" where "XXX" corresponds to the solution number. If the argument to x contains multiple zones, the fields containing solutions are named "solution_XXX_YYY" where "XXX" corresponds to the solution and "YYY" is the name of the management zone.

After solving problems that contain multiple zones, it may be useful to use the category_layer() or category_vector() function to reformat the output.

Value

A numeric, matrix, RasterLayer, Spatial, or sf::sf() object containing the solution to the problem. Additionally, the returned object will have the following additional attributes: "objective" containing the solution's objective, "runtime" denoting the number of seconds that elapsed while solving the problem, and "status" describing the status of the solution (e.g. "OPTIMAL" indicates that the optimal solution was found). In most cases, the first solution (e.g. "solution_001") will contain the best solution found by the solver (note that this may not be an optimal solution depending on the gap used to solve the problem and noting that the default gap is 0.1).

See Also

eval_feature_representation_summary(), problem(), solvers, category_layer(), presolve_check().

Examples

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

# load data
data(sim_pu_raster, sim_pu_polygons, sim_pu_sf, sim_features,
     sim_pu_zones_stack, sim_pu_zones_sf, sim_features_zones)

# build minimal conservation problem with raster data
p1 <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve the problem
s1 <- solve(p1)

# print solution
print(s1)

# print attributes describing the optimization process and the solution
print(attr(s1, "objective"))
```

```
print(attr(s1, "runtime"))
print(attr(s1, "status"))

# calculate feature representation in the solution
r1 <- eval_feature_representation_summary(p1, s1)
print(r1)

# plot solution
plot(s1, main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
# build minimal conservation problem with polygon (Spatial) data
p2 <- problem(sim_pu_polygons, sim_features, cost_column = "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve the problem
s2 <- solve(p2)

# print first six rows of the attribute table
print(head(s2))

# calculate feature representation in the solution
r2 <- eval_feature_representation_summary(p2, s2[, "solution_1"])
print(r2)

# plot solution
spplot(s2, zcol = "solution_1", main = "solution", axes = FALSE, box = FALSE)

## End(Not run)

# build minimal conservation problem with polygon (sf) data
p3 <- problem(sim_pu_sf, sim_features, cost_column = "cost") %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve the problem
s3 <- solve(p3)

# print first six rows of the attribute table
print(head(s3))

# calculate feature representation in the solution
r3 <- eval_feature_representation_summary(p3, s3[, "solution_1"])
print(r3)

# plot solution
plot(s3[, "solution_1"])
```

```
## End(Not run)

# build multi-zone conservation problem with raster data
p4 <- problem(sim_pu_zones_stack, sim_features_zones) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve the problem
s4 <- solve(p4)

# print solution
print(s4)

# calculate feature representation in the solution
r4 <- eval_feature_representation_summary(p4, s4)
print(r4)

# plot solution
plot(category_layer(s4), main = "solution", axes = FALSE, box = FALSE)

## End(Not run)
# build multi-zone conservation problem with polygon (sf) data
p5 <- problem(sim_pu_zones_sf, sim_features_zones,
              cost_column = c("cost_1", "cost_2", "cost_3")) %>%
      add_min_set_objective() %>%
      add_relative_targets(matrix(runif(15, 0.1, 0.2), nrow = 5,
                                  ncol = 3)) %>%
      add_binary_decisions() %>%
      add_default_solver(verbose = FALSE)
## Not run:
# solve the problem
s5 <- solve(p5)

# print first six rows of the attribute table
print(head(s5))

# calculate feature representation in the solution
r5 <- eval_feature_representation_summary(
  p5, s5[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")])
print(r5)

# create new column representing the zone id that each planning unit
# was allocated to in the solution
s5$solution <- category_vector(s5[, c("solution_1_zone_1",
                                      "solution_1_zone_2",
                                      "solution_1_zone_3")])
s5$solution <- factor(s5$solution)

# plot solution
plot(s5[, "solution"])
```

```
## End(Not run)
```

`Solver-class` *Solver prototype*

Description

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

Fields

$name `character` name of solver.

$data `list` object optimization problem data.

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

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

Usage

`x$print()`

`x$show()`

`x$repr()`

`x$get_data(name)`

`x$set_data(name,value)`

`x$set_variable_ub(index,value)`

`x$set_variable_lb(index,value)`

`x$calculate(op)`

`x$run()`

`x$solve(op)`

Arguments

x `Solver` object.

op `OptimizationProblem` object.

Details

print print the object.

show show the object.

repr character representation of object.

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

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

set_variable_ub set the upper bounds on decision variables in a pre-calculated optimization problem stored in the solver.

set_variable_lb set the lower bounds on decision variables in a pre-calculated optimization problem stored in the solver.

calculate ingest a general purpose OptimizationProblem object and convert it to the correct format for the solver.

run run the solver and output a solution

solve solve an OptimizationProblem using this object.

---

solvers *Problem solvers*

Description

Specify the software and configuration used to solve a conservation planning problem(). By default, the best available software currently installed on the system will be used. For information on the performance of different solvers, please see Schuster *et al.* (2020) for benchmarks comparing the run time and solution quality of some of these solvers when applied to different sized datasets.

Details

The following solvers can be used to find solutions for a conservation planning problem():

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

add_gurobi_solver() *Gurobi* is a state-of-the-art commercial optimization software with an R package interface. We recommend using this solver if at all possible. It is by far the fastest of the solvers available for generating prioritizations, however, it is not freely available. That said, licenses are available to academics at no cost. The **gurobi** package is distributed with the *Gurobi* software suite. This solver uses the **gurobi** package to solve problems.

add_cplex_solver() *IBM CPLEX* is a commercial optimization software. It is faster than the open source solvers available for generating prioritizations, however, it is not freely available. Similar to the *Gurobi* software, licenses are available to academics at no cost. This solver uses the **cplexAPI** package to solve problems using *IBM CPLEX*.

`add_cbc_solver()` *CBC* is an open-source mixed integer programming solver that is part of the Computational Infrastructure for Operations Research (COIN-OR) project. Preliminary benchmarks indicate that it is the fastest open source solver currently supported. We recommend using this solver if both *Gurobi* and *IBM CPLEX* are unavailable. It requires the **rcbc** package, which is currently only available on GitHub.

`add_lpsymphony_solver()` *SYMPHONY* is an open-source mixed integer programming solver that is also part of the COIN-OR project. Although both *SYMPHONY* and *CBC* are part of the COIN-OR project, they are different software. The **lpsymphony** package provides an interface to the *SYMPHONY* software, and is distributed through Bioconductor. We recommend using this solver if the CBC solver cannot be installed. This solver can use parallel processing to solve problems, so it is faster than **Rsymphony** package interface (see below).

`add_rsymphony_solver()` This solver provides an alternative interface to the *SYMPHONY* solver using the **Rsymphony** package. Unlike other solvers, the **Rsymphony** package can be installed directly from the Comprehensive R Archive Network (CRAN). It is also the slowest of the available solvers.

References

Schuster R, Hanson JO, Strimas-Mackey M, and Bennett JR (2020). Exact integer linear programming solvers outperform simulated annealing for solving conservation planning problems. *PeerJ*, 8: e9258.

See Also

constraints, decisions, objectives, penalties, portfolios, `problem()`, targets.

Examples

```
## Not run:
# load data
data(sim_pu_raster, sim_features)

# create basic problem
p <- problem(sim_pu_raster, sim_features) %>%
  add_min_set_objective() %>%
  add_relative_targets(0.1) %>%
  add_proportion_decisions()

# create vector to store plot titles
titles <- c()

# create empty stack to store solutions
s <- stack()

# if gurobi is installed: create problem with added gurobi solver
if (require("gurobi")) {
  titles <- c(titles, "gurobi")
  p3 <- p %>% add_gurobi_solver(verbose = FALSE)
  s <- addLayer(s, solve(p3))
}
```

```
# if cplexAPI is installed: create problem with added CPLEX solver
if (require("cplexAPI")) {
  titles <- c(titles, "CPLEX")
  p4 <- p %>% add_cplex_solver(verbose = FALSE)
  s <- addLayer(s, solve(p4))
}

# if rcbc is installed: create problem with added cbc solver
if (require("rcbc")) {
  titles <- c(titles, "CBC")
  p6 <- p %>% add_cbc_solver(verbose = FALSE)
  s <- addLayer(s, solve(p6))
}

# create problem with added rsymphony solver
if (require("Rsymphony")) {
  titles <- c(titles, "Rsymphony")
  p2 <- p %>% add_rsymphony_solver(verbose = FALSE)
  s <- addLayer(s, solve(p2))
}

# if lpsymphony is installed: create problem with added lpsymphony solver
if (require("lpsymphony")) {
  titles <- c(titles, "lpsymphony")
  p5 <- p %>% add_lpsymphony_solver(verbose = FALSE)
  s <- addLayer(s, solve(p5))
}

# plot solutions
plot(s, main = titles, axes = FALSE, box = FALSE)

## End(Not run)
```

---

summaries *Evaluate solutions using summary statistics*

Description

After generating a solution to a conservation planning `problem()`, it can be useful to evaluate how well it performs. These functions can be used to evaluate a solution according to various different summary statistics.

Details

The following functions can be used to summarize the performance of a solution to a conservation planning `problem()`:

`eval_n_summary()` Calculate the number of planning units selected within a solution.

eval_cost_summary() Calculate the total cost of a solution.

eval_feature_representation_summary() Calculate how well features are represented by a solution. This function can be used for all problems.

eval_target_coverage_summary() Calculate how well feature representation targets are met by a solution. This function can only be used with problems contain targets.

eval_boundary_summary() Calculate the exposed boundary length (perimeter) associated with a solution.

eval_connectivity_summary() Calculate the connectivity held within a solution.

See Also

constraints, decisions, objectives penalties, portfolios, problem(), solvers.

Examples

```
# load data
data(sim_pu_raster, sim_features)

# create a minimal problem
p <- problem(sim_pu_raster, sim_features) %>%
     add_min_set_objective() %>%
     add_relative_targets(0.1) %>%
     add_binary_decisions() %>%
     add_default_solver(verbose = FALSE)

## Not run:
# solve problem
s <- solve(p)

# evaluate number of selected planning units in solution
eval_n_summary(p, s)

# evaluate solution cost
eval_cost_summary(p, s)

# evaluate feature representation by solution
eval_feature_representation_summary(p, s)

# evaluate target coverage by solution
eval_target_coverage_summary(p, s)


# evaluate exposed boundary (perimeter) length by solution
eval_boundary_summary(p, s)

# create a connectivity matrix to describe pair-wise connectivity
# values between combinations of planning units,
# see ?connectivity_matrix for more information

# for brevity, we will do this using the cost data
# cost valuers have high connectivity between them
```

```
cm <- connectivity_matrix(sim_pu_raster, sim_pu_raster)

# evaluate connectivity of solution
eval_connectivity_summary(p, s, data = cm)


## End(Not run)
```

Target-class *Target prototype*

Description

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

See Also

`ConservationModifier`.

---

targets *Add representation targets*

Description

Targets are used to specify the minimum amount or proportion of a feature's distribution that should (ideally) be covered (represented) by a solution.

Details

Please note that most objectives require targets, and attempting to solve a problem that requires targets will throw an error.

The following functions can be used to specify targets for a conservation planning `problem()`:

`add_relative_targets()` Set targets as a proportion (between 0 and 1) of the total amount of each feature in the the study area.

`add_absolute_targets()` Set targets that denote the minimum amount of each feature required in the prioritization.

`add_loglinear_targets()` Set targets as a proportion (between 0 and 1) that are calculated using log-linear interpolation.

`add_manual_targets()` Set targets manually.

See Also

constraints, decisions, objectives penalties, portfolios, `problem()`, solvers.

Examples

```
# load data
data(sim_pu_raster, sim_features)

# create base problem
p <- problem(sim_pu_raster, sim_features) %>%
     add_min_set_objective() %>%
     add_binary_decisions() %>%
     add_default_solver(verbose = FALSE)

# create problem with added relative targets
p1 <- p %>% add_relative_targets(0.1)

# create problem with added absolute targets
p2 <- p %>% add_absolute_targets(3)

# create problem with added loglinear targets
p3 <- p %>% add_loglinear_targets(10, 0.9, 100, 0.2)

# create problem with manual targets that equate to 10% relative targets
p4 <- p %>% add_manual_targets(data.frame(feature = names(sim_features),
                                          target = 0.1,
                                          type = "relative"))
## Not run:
# solve problem
s <- stack(solve(p1), solve(p2), solve(p3), solve(p4))

# plot solution
plot(s, axes = FALSE, box = FALSE,
     main = c("relative targets", "absolute targets", "loglinear targets",
              "manual targets"))

## End(Not run)
```

---

tibble-methods *Manipulate tibbles*

Description

Assorted functions for manipulating `tibble::tibble()` objects.

Usage

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

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

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

Arguments

x	`tibble::tibble()` object.

Details

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

nrow extract `integer` number of rows.

ncol extract `integer` number of columns.

as.list convert to a `list`.

print print the object.

Examples

```
# load tibble package
require(tibble)

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

# number of rows
nrow(a)

# number of columns
ncol(a)

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

`write_problem` *Write problem*

Description

Save the mathematical formulation for a conservation planning `problem()` to a file for mixed integer programming solvers. Note that this function requires the **Rsymphony** package to be installed.

Usage

```
write_problem(x, path)
```

Arguments

x — problem() (i.e. ConservationProblem) object.

path — character file path to save the problem formulation. The argument should contain a ".lp" or .mps" file extension to specify whether the problem formulation will be saved in the LP or MPS format (respectively).

Value

Invisible TRUE indicating success.

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
data(sim_pu_raster, sim_features, sim_pu_zones_stack, sim_features_zones)

# create minimal problem
p <- problem(sim_pu_raster, sim_features) %>%
      add_min_set_objective() %>%
      add_relative_targets(0.1) %>%
      add_binary_decisions()

# save problem to file (using the Rsymphony R package)
write_problem(p, "model.lp")

## End(Not run)
```

---

zones — *Management zones*

Description

Organize biodiversity data into the expected amount of different features under different management zones.

Usage

```
zones(..., zone_names = NULL, feature_names = NULL)
```

Arguments

... — raster::raster() or character objects that pertain to the biodiversity data. See Details for more information.

zone_names	character names of the management zones. Defaults to NULL which results in sequential integers.
feature_names	character names of the features zones. Defaults to NULL which results in sequential integers.

Details

This function is used to store and organize data for use in a conservation planning problem() that has multiple management zones. In all cases, the data for each zone is input as a separate argument. The correct arguments depends on the type of planning unit data used when building the conservation planning problem().

Raster, Spatial Raster data denoting the amount of each feature present assuming each management zone. Data for each zone are specified in separate arguments, and the data for each feature in a given zone are specified in separate layers in a raster::stack() object. Note that all layers for a given zone must have NA values in exactly the same cells.

Spatial(), data.frame character vector with column names that correspond to the abundance or occurrence of different features in each planning unit for each zone. Note that these columns must not contain any NA values.

Spatial(), data.frame **or** matrix data.frame denoting the amount of each feature in each zone. Following conventions used in *Marxan*, data.frame objects should be supplied with the columns:

pu integer planning unit identifier.

species integer feature identifier.

amount numeric amount of the feature in the planning unit for a given zone.

Note that data for each zone are specified in a separate argument, and the data contained in a single data.frame object correspond to a single zone. Also, note that data are not required for all combinations of planning units, features, and zones. The amounts of features in planning units assuming different management zones that are missing from the table are treated as zero.

Value

Zones object.

See Also

problem().

Examples

```
# load planning unit data
data(sim_pu_raster)

zone_1 <- simulate_species(sim_pu_raster, 3)
zone_2 <- simulate_species(sim_pu_raster, 3)

# create zones using two raster stack objects
# each object corresponds to a different zone and each layer corresponds to
```

```
# a different species
z <- zones(zone_1, zone_2, zone_names = c("zone_1", "zone_2"),
           feature_names = c("feature_1", "feature_2", "feature_3"))
print(z)

# note that the do.call function can also be used to create a Zones object
# this method for creating a Zones object can be helpful when there are many
# management zones
l <- list(zone_1, zone_2, zone_names = c("zone_1", "zone_2"),
          feature_names = c("feature_1", "feature_2", "feature_3"))
z <- do.call(zones, l)
print(z)

# create zones using character vectors that represent the names of
# fields (columns) in a data.frame or Spatial object that contain the amount
# of each species expected different management zones
z <- zones(c("spp1_zone1", "spp2_zone1"),
           c("spp1_zone2", "spp2_zone2"),
           c("spp1_zone3", "spp2_zone3"),
           zone_names = c("zone1", "zone2", "zone3"),
           feature_names = c("spp1", "spp2"))
print(z)
```

---

zone_names *Zone names*

Description

Extract the names of zones in an object.

Usage

```
zone_names(x)

## S4 method for signature 'ConservationProblem'
zone_names(x)

## S4 method for signature 'ZonesRaster'
zone_names(x)

## S4 method for signature 'ZonesCharacter'
zone_names(x)
```

Arguments

x	problem() (i.e. ConservationProblem) or Zones()

Value

character zone names.

Examples

```
# load data
data(sim_pu_zones_stack, sim_features_zones)

# print names of zones in a Zones object
print(zone_names(sim_features_zones))
# create problem with multiple zones
p <- problem(sim_pu_zones_stack, sim_features_zones) %>%
     add_min_set_objective() %>%
     add_relative_targets(matrix(0.2, ncol = 3, nrow = 5)) %>%
     add_binary_decisions()

# print zone names in problem
print(zone_names(p))
```

`%>%` *Pipe operator*

Description

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

Arguments

`lhs, rhs`	An object and a function.

Value

An object.

See Also

`magrittr::%>%()`, `tee()`.

Examples

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

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

# reset the seed
set.seed(500)

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

%T>% *Tee operator*

Description

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

Arguments

lhs, rhs	An object and a function.

Value

An object.

See Also

magrittr::%T>%(), pipe().

Examples

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

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

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

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

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

Index