Package 'prioritizr'

November 10, 2025

Type Package

Version 8.1.0

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). Users can also install the IBM CPLEX software (<https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-optimizer>) and the 'cplexAPI' R package (available at <https://github.com/cran/cplexAPI>). Additionally, the 'rcbc' R package (available at <https://github.com/dirkschumacher/rcbc>) can be used to generate solutions using the CBC optimization software (<https://github.com/coin-or/Cbc>). For further details, see Hanson et al. (2025) <doi:10.1111/cobi.14376>.

Imports utils, methods, parallel, R6 (>= 2.5.1), rlang (>= 1.1.0), cli (>= 3.6.0), sf (>= 1.0-12), units (>= 0.8.7), terra (>= 1.8-54), raster (>= 3.6.11), Matrix (>= 1.3-0), assertthat (>= 0.2.0), igraph (>= 2.0.3), ape (>= 5.6-1), magrittr (>= 2.0.1), exactextractr (>= 0.8.1), tibble (>= 2.0.0), withr (>= 2.3.0)

Suggests testthat (>= 3.1.0), knitr (>= 1.36), gurobi (>= 8.0-1), rcbc (>= 0.1.0.9001), cplexAPI (>= 1.4.0), lpsymphony (>= 1.17.0), slam (>= 0.1-48), Rsymphony (>= 0.1-31), highs (>= 0.1-10), rmarkdown (>= 2.11), prioritizrdata (>= 0.2.4), fields (>= 14.0), vroom (>= 1.6.5)

Depends R (>= 4.1.0)

LinkingTo Rcpp (>= 1.0.7), RcppArmadillo (>= 0.10.7.3.0), BH (>= 1.75.0-0)

License GPL-3

Language en-US

Encoding UTF-8

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

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

VignetteBuilder knitr

RoxygenNote 7.3.2

Collate 'external-classes.R' 'internal.R' 'waiver.R'
'ConservationModifier-class.R' 'category_vector.R'
'category_layer.R' 'binary_stack.R'
'ConservationProblem-class.R' 'Constraint-class.R'
'Decision-class.R' 'Objective-class.R'
'OptimizationProblem-class.R' 'reexports.R'
'OptimizationProblem-methods.R' 'Penalty-class.R'
'Portfolio-class.R' 'RcppExports.R' 'Solver-class.R'
'Target-class.R' 'TargetMethod-class.R' 'Weight-class.R'
'zones.R' 'add_absolute_targets.R'
'marxan_connectivity_data_to_matrix.R'
'add_asym_connectivity_penalties.R' 'get_target_method.R'
'target_optimization_format.R' 'add_auto_targets.R'
'add_binary_decisions.R' 'marxan_boundary_data_to_matrix.R'
'add_boundary_penalties.R' 'add_cbc_solver.R'
'add_compile_solver.R' 'add_connectivity_penalties.R'
'add_contiguity_constraints.R' 'add_cplex_solver.R'
'add_cuts_portfolio.R' 'add_default_portfolio.R'
'standalone-assertions_misc.R' 'add_default_solver.R'
'add_extra_portfolio.R' 'add_feature_contiguity_constraints.R'
'add_feature_weights.R' 'add_gap_portfolio.R'
'add_group_targets.R' 'add_gurobi_solver.R'
'add_highs_solver.R' 'add_linear_constraints.R'
'add_linear_penalties.R' 'intersecting_units.R'
'add_locked_in_constraints.R' 'add_locked_out_constraints.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_penalties_objective.R' 'add_min_set_objective.R'
'add_min_shortfall_objective.R' 'add_neighbor_constraints.R'
'add_neighbor_penalties.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' 'as.R' 'as_units.R'
'boundary_matrix.R' 'branch_matrix.R'
'calibrate_cohon_penalty.R' 'compile.R' 'connectivity_matrix.R'
'constraints.R' 'data.R' 'decisions.R' 'deprecated.R'
'eval_asym_connectivity_summary.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'
'presolve_check.R' 'solve.R' 'problem.R'
'eval_rank_importance.R' 'eval_rare_richness_importance.R'
'planning_unit_solution_format.R'
'eval_replacement_importance.R'
'eval_target_coverage_summary.R' 'fast_extract.R'
'feature_abundances.R' 'feature_names.R' 'importance.R'
'knit_print.R' 'linear_interpolation.R'
'loglinear_interpolation.R' 'read_marxan_data.R'
'read_marxan_parameters.R' 'marxan_problem.R' 'memory.R'
'number_of_features.R' 'number_of_planning_units.R'
'number_of_total_units.R' 'number_of_zones.R' 'objectives.R'
'optimization_problem.R' 'package.R' 'penalties.R'
'planning_unit_indices.R' 'planning_unit_solution_status.R'
'portfolios.R' 'print.R' 'problem-deprecated.R'
'proximity_matrix.R' 'rescale_matrix.R' 'rij_matrix.R'
'run_calculations.R' 'show.R' 'simulate.R' 'solvers.R'
'spec_absolute_targets.R' 'spec_area_targets.R'
'spec_duran_targets.R' 'spec_interp_absolute_targets.R'
'spec_interp_area_targets.R' 'spec_jung_targets.R'
'spec_max_targets.R' 'spec_min_targets.R'
'spec_polak_targets.R' 'spec_pop_size_targets.R'
'spec_relative_targets.R' 'spec_rl_ecosystem_targets.R'
'spec_rl_species_targets.R' 'spec_rodrigues_targets.R'
'spec_rule_targets.R' 'spec_ward_targets.R'
'spec_watson_targets.R' 'spec_wilson_targets.R'
'standalone-all_binary.R' 'standalone-all_columns_any_finite.R'
'standalone-all_columns_inherit.R' 'standalone-all_finite.R'
'standalone-all_positive.R' 'standalone-all_proportion.R'
'standalone-any_nonNA.R' 'standalone-any_nonzero.R'
'standalone-as_Matrix.R'
'standalone-assertions_ConservationProblem.R'
'standalone-assertions_class.R'
'standalone-assertions_functions.R'
'standalone-assertions_handlers.R'
'standalone-assertions_raster.R' 'standalone-assertions_sf.R'
'standalone-assertions_solvers.R'
'standalone-assertions_vector.R' 'standalone-cli.R'
'standalone-get_crs.R' 'standalone-is_numeric_values.R'
'standalone-is_same_crs.R'

'standalone-is_spatial_extents_overlap.R' 'standalone-repr.R' 'summaries.R' 'summary.R' 'targets.R' 'test-helpers.R' 'validate_marxan_data.R' 'write_problem.R' 'zone_names.R' 'zzz.R'

Config/testthat/edition 3

Config/Needs/website topsis, vegan, cluster, units, scales, dplyr, ggplot2, piggyback, prioritizrdata, pkgdown

NeedsCompilation yes

Author Jeffrey O Hanson [aut] (ORCID: <https://orcid.org/0000-0002-4716-6134>), Richard Schuster [aut, cre] (ORCID: <https://orcid.org/0000-0003-3191-7869>), Nina Morrell [aut], Matthew Strimas-Mackey [aut] (ORCID: <https://orcid.org/0000-0001-8929-7776>), Brandon P M Edwards [aut] (ORCID: <https://orcid.org/0000-0003-0865-3076>), Matthew E Watts [aut], Peter Arcese [aut] (ORCID: <https://orcid.org/0000-0002-8097-482X>), Joseph R Bennett [aut] (ORCID: <https://orcid.org/0000-0002-3901-9513>), Hugh P Possingham [aut] (ORCID: <https://orcid.org/0000-0001-7755-996X>)

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

Repository CRAN

Date/Publication 2025-11-10 13:20:08 UTC

Contents

`add_absolute_targets` *Add absolute targets*

Description

Add targets to a conservation planning problem expressed as the same values as the underlying feature data (ignoring any specified feature units). For example, setting a target of 10 for a feature specifies that a solution should ideally select a set of planning units that contain a total (summed) value of, at least, 10 for the feature.

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() object.
targets	object that specifies the targets for each feature. See the Targets format section for more information.

Details

This function is used to set targets for each feature (separately). For problems associated with a single management zone, this function may be useful to specify individual targets for each feature. For problems associated with multiple management zones, this function can also be used to specify a target for each feature within each zone (separately). For example, this may be useful in planning exercises where it is important to ensure that some of the features are adequately represented by multiple zones. For example, in a marine spatial planning exercise, it may be important for some features (e.g., commercial important fish species) to be adequately represented by a conservation zone for ensuring their long-term persistence, and also by a fishing zone to for ensure food security. For greater flexibility in target setting (such as setting targets that can be met through the allocation of multiple zones), see the add_manual_targets() function.

Value

An updated problem() object with the targets added to it.

Targets format

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

targets **as a** numeric **vector** containing target values for each feature. Additionally, for convenience, this format can be a single value to assign the same target to each feature. Note that this format cannot be used to specify targets for problems with multiple zones.

targets **as a** matrix **object** 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.

targets **as a** character **vector** containing the column name(s) in the feature data associated with the argument to x that contain targets. This format can only be used when the feature data associated with x is a sf::st_sf() or data.frame. For problems that contain a single zone, the argument to targets must contain a single column name. Otherwise, for problems that contain multiple zones, the argument to targets must contain a column name for each zone.

Target setting

Many conservation planning problems require targets. Targets are used to specify the minimum amount, or proportion, of a feature's spatial distribution that should ideally be protected. This is important so that the optimization process can weigh the merits and trade-offs between improving

the representation of one feature over another feature. Although it can be challenging to set meaningful targets, this is a critical step for ensuring that prioritizations meet the stakeholder objectives that underpin a prioritization exercise (Carwardine *et al.* 2009). In other words, targets play an important role in ensuring that a priority setting process is properly tuned according to stakeholder requirements. For example, targets provide a mechanism for ensuring that a prioritization secures enough habitat to promote the long-term persistence of each threatened species, culturally important species, or economically important ecosystem services under consideration. Since there is often uncertainty regarding stakeholder objectives (e.g., how much habitat should be protected for a given species) or the influence of particular target on a prioritization (e.g., how would setting a 90% or 100% for a threatened species alter priorities), it is often useful to generate and compare a suite of prioritizations based on different target scenarios.

References

Carwardine J, Klein CJ, Wilson KA, Pressey RL, Possingham HP (2009) Hitting the target and missing the point: target-based conservation planning in context. *Conservation Letters*, 2: 4–11.

See Also

Other functions for adding targets: `add_auto_targets()`, `add_group_targets()`, `add_manual_targets()`, `add_relative_targets()`

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# create minimal problem with no targets
p0 <-
  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 <- p0 %>% add_absolute_targets(3)

# create problem with varying targets for each feature
targets <- c(1, 2, 3, 2, 1)
p2 <- p0 %>% add_absolute_targets(targets)

# solve problem
s1 <- c(solve(p1), solve(p2))
names(s1) <- c("equal targets", "varying targets")
```

```
# plot solution
plot(s1, axes = FALSE)

# create a problem with multiple management zones
p3 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  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_zones_features),
  ncol = number_of_zones(sim_zones_features),
  dimnames = list(
    feature_names(sim_zones_features), zone_names(sim_zones_features)
  )
)
print(p4_targets)

p4 <- p3 %>% add_absolute_targets(p4_targets)

# solve problem
s4 <- solve(p4)

# plot solution (cell values correspond to zone identifiers)
plot(category_layer(s4), main = "equal targets", axes = FALSE)

# 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_zones_features),
  ncol = number_of_zones(sim_zones_features),
  dimnames = list(
    feature_names(sim_zones_features),
    zone_names(sim_zones_features)
  )
)
print(p5_targets)

p5 <- p3 %>% add_absolute_targets(p5_targets)

# solve problem
s5 <- solve(p5)

# plot solution (cell values correspond to zone identifiers)
plot(category_layer(s5), main = "varying targets", axes = FALSE)

## End(Not run)
```

add_asym_connectivity_penalties
Add asymmetric connectivity penalties

Description

Add penalties to a conservation planning problem to account for asymmetric connectivity between planning units. Asymmetric connectivity data describe connectivity information that is directional. For example, asymmetric connectivity data could describe the strength of rivers flowing between different planning units. Since river flow is directional, the level of connectivity from an upstream planning unit to a downstream planning unit would be higher than that from a downstream planning unit to an upstream planning unit.

Usage

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

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

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

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

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

Arguments

x	problem() 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 working with multiple zones and the argument to `data` is a `matrix` or `Matrix` object. If the argument to `data` is an `array` or `data.frame` with data for multiple zones (e.g., using the `"zone1"` and `"zone2"` column names), 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 penalizes solutions that select planning units that share high connectivity values with other planning units that are not selected by the solution (based on Beger *et al.* 2010).

Value

An updated `problem()` object with the penalties added to it.

Mathematical formulation

The connectivity penalties are implemented 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:

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

Otherwise, if the argument to `data` is supplied as an `array` object, then the penalties are calculated as:

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

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 $-p$. Additionally, to linearize the problem, the expression $X_{iz} \times X_{jy}$ is modeled using a set of continuous variables (bounded between 0 and 1) based on Beyer *et al.* (2016).

Data format

The argument to `data` can be specified using several different formats.

`data` **as a** `matrix`/`Matrix` **object** 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` **as a** `data.frame` **object** containing columns that are named `"id1"`, `"id2"`, and `"boundary"`. Here, each row denotes the connectivity between a pair of planning units (per values in the `"id1"` and `"id2"` columns) following the *Marxan* format. If the argument to `x` contains multiple zones, then the `"zone1"` and `"zone2"` columns can optionally be provided to manually specify the connectivity values between planning units when they are allocated to specific zones. If the `"zone1"` and `"zone2"` columns are present, then the argument to `zones` must be `NULL`.

`data` **as an** `array` **object** 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.

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

See penalties for an overview of all functions for adding penalties. Also see `calibrate_cohon_penalty()` for assistance with selecting an appropriate `penalty` value.

Other functions for adding penalties: `add_boundary_penalties()`, `add_connectivity_penalties()`, `add_feature_weights()`, `add_linear_penalties()`, `add_neighbor_penalties()`

Examples

```
## Not run:
```

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

# load data
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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 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.
acm1 <- matrix(0, nrow(sim_pu_polygons), nrow(sim_pu_polygons))
acm1 <- as(acm1, "Matrix")
centroids <- sf::st_coordinates(
  suppressWarnings(sf::st_centroid(sim_pu_polygons))
)
adjacent_units <- sf::st_intersects(sim_pu_polygons, sparse = FALSE)
for (i in seq_len(nrow(sim_pu_polygons))) {
  for (j in seq_len(nrow(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[i, 1] == centroids[j, 1]) {
        if (centroids[i, 2] > centroids[j, 2]) {
          # if i is north of j add 10 units of connectivity
          acm1[i, j] <- acm1[i, j] + 10
        } else if (centroids[i, 2] < centroids[j, 2]) {
          # if i is south of j add 1 unit of connectivity
          acm1[i, j] <- acm1[i, j] + 1
        }
      }
    }
  }
}

# rescale matrix values to have a maximum value of 1
acm1 <- rescale_matrix(acm1, max = 1)

# visualize asymmetric connectivity matrix
Matrix::image(acm1)

# create penalties
penalties <- c(1, 50)
```

```
# create problems using the different penalties
p2 <- list(
  p1,
  p1 %>% add_asym_connectivity_penalties(penalties[1], data = acm1),
  p1 %>% add_asym_connectivity_penalties(penalties[2], data = acm1)
)

# solve problems
s2 <- lapply(p2, solve)

# create object with all solutions
s2 <- sf::st_sf(
  tibble::tibble(
    p2_1 = s2[[1]]$solution_1,
    p2_2 = s2[[2]]$solution_1,
    p2_3 = s2[[3]]$solution_1
 ),
 geometry = sf::st_geometry(s2[[1]])
)

names(s2)[1:3] <- c("basic problem", paste0("acm1 (", penalties,")"))

# plot solutions based on different penalty values
plot(s2, cex = 1.5)

# create minimal multi-zone problem and limit solver to one minute
# to obtain solutions in a short period of time
p3 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  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)

# crate asymmetric connectivity data by randomly simulating values
acm2 <- matrix(
  runif(ncell(sim_zones_pu_raster) ^ 2),
  nrow = ncell(sim_zones_pu_raster)
)

# create multi-zone problems using the penalties
p4 <- list(
  p3,
  p3 %>% add_asym_connectivity_penalties(penalties[1], data = acm2),
  p3 %>% add_asym_connectivity_penalties(penalties[2], data = acm2)
)

# solve problems
s4 <- lapply(p4, solve)
s4 <- lapply(s4, category_layer)
s4 <- terra::rast(s4)
names(s4) <- c("basic problem", paste0("acm2 (", penalties,")"))
```

```
# plot solutions
plot(s4, axes = FALSE)

## End(Not run)
```

`add_auto_targets` *Add targets automatically*

Description

Add targets to a conservation planning problem based on a particular target setting method.

Usage

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

## S4 method for signature 'ConservationProblem,list'
add_auto_targets(x, method)

## S4 method for signature 'ConservationProblem,TargetMethod'
add_auto_targets(x, method)
```

Arguments

`x`	`problem()` object.
`method`	`character` value, `character` vector, `list`, or object (`TargetMethod`) specifying the target setting method. See Target methods below for more information.

Value

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

Target methods

A variety of options are available for specifying target setting methods.

`method` **is a** `character` **value** This option involves specifying a target setting method based on its name. It is particularly useful when all features should be assigned targets based on the same method, and the method should rely on default parameters. For example, if `x` was a `problem()`, then the following code could be used to specify that all features should have their targets calculated based on Jung *et al.* (2021) with default parameters.

```
x %>% add_auto_targets(method = "jung"))
```

The following character values can be used to specify target setting methods: "jung" (per Jung *et al.* 2021), "polak" (per Polak *et al.* 2016), "rodrigues" (per Rodrigues *et al.* 2004), "ward" (per Ward *et al.* 2025), and "watson" (per Watson *et al.* 2010). (2010). Additionally, the following values can be used to set targets based on criteria from the IUCN Red List of Threatened Species (IUCN 2025): "rl_species_VU_A1_B1" "rl_species_EN_A1_B1", "rl_species_CR_A1_B1", "rl_species_VU_A1_B2" "rl_species_EN_A1_B2", "rl_species_CR_A1_B2", "rl_species_VU_A2_B1" "rl_species_EN_A2_B1", "rl_species_CR_A2_B1", "rl_species_VU_A2_B2" "rl_species_EN_A2_B2", "rl_species_CR_A2_B2", "rl_species_VU_A3_B1" "rl_species_EN_A3_B1", "rl_species_CR_A3_B1", "rl_species_VU_A3_B2" "rl_species_EN_A3_B2", "rl_species_CR_A3_B2", "rl_species_VU_A4_B1" "rl_species_EN_A4_B1", "rl_species_CR_A4_B1", "rl_species_VU_A4_B2" "rl_species_EN_A4_B2", and "rl_species_CR_A4_B2". Furthermore, the following values can be used to set targets based on criteria from the IUCN Red List of Ecosystems (IUCN 2024): "rl_ecosystem_VU_A1_B1", "rl_ecosystem_EN_A1_B1", "rl_ecosystem_CR_A1_B1", "rl_ecosystem_VU_A1_B2", "rl_ecosystem_EN_A1_B2", "rl_ecosystem_CR_A1_B2", "rl_ecosystem_VU_a2a_B1 "rl_ecosystem_EN_a2a_B1", "rl_ecosystem_CR_a2a_B1", "rl_ecosystem_VU_a2a_B2", "rl_ecosystem_EN_a2a_B2", "rl_ecosystem_CR_a2a_B2", "rl_ecosystem_VU_a2b_B1", "rl_ecosystem_EN_a2b_B1", "rl_ecosystem_CR_a2b_B1", "rl_ecosystem_VU_a2b_B2", "rl_ecosystem_EN_a2b_B2", "rl_ecosystem_CR_a2b_B2", "rl_ecosystem_VU_A3_B1", "rl_ecosystem_EN_A3_B "rl_ecosystem_CR_A3_B1", "rl_ecosystem_VU_A3_B2", "rl_ecosystem_EN_A3_B2", and "rl_ecosystem_CR_A3_B2".' For convenience, these options can also be specified with lower case letters (e.g., "rl_species_VU_A1_B1" may alternatively be specified as "rl_species_vu_a1_b1"). Note that target setting methods that require additional data or parameters cannot be specified with character values.

method **is a** character **vector** This option involves specifying a target setting method for each feature based on the name of the method. It is particularly useful when particular features should be assigned targets based on different methods, and all methods rely on default parameters. For example, if x was a problem() with three features, then the following code could be used to specify a target based on Jung *et al.* (2021) for the first feature, target based on Polak *et al.* (2015) for the second feature, and target based on Jung *et al.* (2021) for the third feature (note that all targets are based on default parameters).

```
x %>% add_auto_targets(method = c("jung", "polak", "jung"))
```

Note that method must specify a value for each feature in x, and the order of the method values should follow the order of the features in x (i.e, per feature_names(x)). For details on the available method values, please refer to the first option where method is a character value.

method **is an object for specifying a method** This option involves specifying a target setting method based on an object. It is particularly useful when all features should be assigned targets based on the same method, and the parameters for calculating the targets should be customized. For example, if x was a problem(), then the following could could be used to specify that all features should have their targets calculated based on Jung *et al.* (2021) with an uplift of 5%.

```
x %>% add_auto_targets(method = jung_targets(prop_uplift = 0.05))
```

The following functions can be used to create an object for specifying a target setting method: spec_absolute_targets(), spec_relative_targets(), spec_area_targets(), spec_interp_absolute_target spec_interp_area_targets(), spec_duran_targets(), spec_jung_targets(), spec_polak_targets(), spec_pop_size_targets(), spec_rl_ecosystem_targets(), spec_rl_species_targets(), spec_rodrigues_targets(), spec_rule_targets(), spec_ward_targets(), spec_watson_targets(), spec_wilson_targets(), spec_min_targets(), and spec_max_targets().

`method` **is a** `list` This option involves specify a target setting method based on a `list` of objects. It is particularly useful when particular features should be assigned different targets based on different methods, and at least one of the methods requires customized parameters. For example, if `x` was a `problem()` with three features, then the following code could be used to specify (i) a target for the first feature based on Jung *et al.* (2021) with customized parameters (i.e, `jung_targets(prop_uplift = 0.05)`), (ii) a target for the second feature based on Polak *et al.* (2015) with default parameters via the name method name (i.e., `"polak"`), and (iii) a target based on Jung *et al.* (2021) with default parameters via an object (i.e., `jung_targets()`).

```
x %>% add_auto_targets(
  method = list(jung_targets(prop_uplift = 0.05), "polak", jung_targets())
)
```

Note that `method` must specify a value for each feature in `x`, and the order of the method values should follow the order of the features in `x` (i.e, per `feature_names(x)`). For details on the available `method` values, please refer to the previous options.

Data calculations

Many of the functions for specifying target setting methods involve calculating targets based on the spatial extent of the features in `x` (e.g., `spec_jung_targets()`, [spec_rodrigues_targets(), and others). Although this function for adding targets can be readily applied to `problem()` objects that have the feature data provided as a `terra::rast()` object, you will need to specify the spatial units for the features when building a `problem()` object if the feature data are provided in a different format. In particular, if the feature data are provided as a `data.frame` or `character` vector, then you will need to specify an argument to `feature_units` when using the `problem()` function. See the Examples section below for a demonstration of using the `feature_units` parameter.

Target setting

Many conservation planning problems require targets. Targets are used to specify the minimum amount, or proportion, of a feature's spatial distribution that should ideally be protected. This is important so that the optimization process can weigh the merits and trade-offs between improving the representation of one feature over another feature. Although it can be challenging to set meaningful targets, this is a critical step for ensuring that prioritizations meet the stakeholder objectives that underpin a prioritization exercise (Carwardine *et al.* 2009). In other words, targets play an important role in ensuring that a priority setting process is properly tuned according to stakeholder requirements. For example, targets provide a mechanism for ensuring that a prioritization secures enough habitat to promote the long-term persistence of each threatened species, culturally important species, or economically important ecosystem services under consideration. Since there is often uncertainty regarding stakeholder objectives (e.g., how much habitat should be protected for a given species) or the influence of particular target on a prioritization (e.g., how would setting a 90% or 100% for a threatened species alter priorities), it is often useful to generate and compare a suite of prioritizations based on different target scenarios.

References

Carwardine J, Klein CJ, Wilson KA, Pressey RL, Possingham HP (2009) Hitting the target and missing the point: target-based conservation planning in context. *Conservation Letters*, 2: 4–11.

Jung M, Arnell A, de Lamo X, García-Rangel S, Lewis M, Mark J, Merow C, Miles L, Ondo I, Pironon S, Ravilious C, Rivers M, Schepaschenko D, Tallowin O, van Soesbergen A, Govaerts R, Boyle BL, Enquist BJ, Feng X, Gallagher R, Maitner B, Meiri S, Mulligan M, Ofer G, Roll U, Hanson JO, Jetz W, Di Marco M, McGowan J, Rinnan DS, Sachs JD, Lesiv M, Adams VM, Andrew SC, Burger JR, Hannah L, Marquet PA, McCarthy JK, Morueta-Holme N, Newman EA, Park DS, Roehrdanz PR, Svenning J-C, Violle C, Wieringa JJ, Wynne G, Fritz S, Strassburg BBN, Obersteiner M, Kapos V, Burgess N, Schmidt- Traub G, Visconti P (2021) Areas of global importance for conserving terrestrial biodiversity, carbon and water. *Nature Ecology and Evolution*, 5:1499–1509.

IUCN (2025) The IUCN Red List of Threatened Species. Version 2025-1. Available at https://www.iucnredlist.org. Accessed on 23 July 2025.

IUCN (2024). Guidelines for the application of IUCN Red List of Ecosystems Categories and Criteria, Version 2.0. Keith DA, Ferrer-Paris JR, Ghoraba SMM, Henriksen S, Monyeki M, Murray NJ, Nicholson E, Rowland J, Skowno A, Slingsby JA, Storeng AB, Valderrábano M, Zager I (Eds.). Gland, Switzerland: IUCN.

Polak T, Watson JEM, Fuller RA, Joseph LN, Martin TG, Possingham HP, Venter O, Carwardine J (2015) Efficient expansion of global protected areas requires simultaneous planning for species and ecosystems. *Royal Society Open Science*, 2: 150107.

Ward M, Possingham HP, Wintle BA, Woinarski JCZ, Marsh JR, Chapple DG, Lintermans M, Scheele BC, Whiterod NS, Hoskin CJ, Aska B, Yong C, Tulloch A, Stewart R, Watson JEM (2025) The estimated cost of preventing extinction and progressing recovery for Australia's priority threatened species. *Proceedings of the National Academy of Sciences*, 122: e2414985122.

Watson JEM, Evans MC, Carwardine J, Fuller RA, Joseph LN, Segan DB, Taylor MFJ, Fensham RJ, Possingham HP (2010) The capacity of Australia's protected-area system to represent threatened species. *Conservation Biology*,25: 324–332.

See Also

Other functions for adding targets: `add_absolute_targets()`, `add_group_targets()`, `add_manual_targets()`, `add_relative_targets()`

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_complex_pu_raster <- get_sim_complex_pu_raster()
sim_complex_features <- get_sim_complex_features()
sim_pu_polygons <- get_sim_pu_polygons()

# create base problem
p0 <-
  problem(sim_complex_pu_raster, sim_complex_features) %>%
  add_min_set_objective() %>%
  add_binary_decisions() %>%
  add_default_solver(gap = 0, verbose = FALSE)

# create problem with Polak et al. (2015) targets by using a function to
```

```
# specify the target setting method
p1 <-
  p0 %>%
  add_auto_targets(method = spec_polak_targets())

# create problem with Polak et al. (2015) targets by using a character value
# to specify the target setting method
# (note that this yields exactly the same targets as p1, and simply
# offers an alternative for specifying targets with default parameters)
p2 <-
  p0 %>%
  add_auto_targets(method = "polak")

# create problem with modified Polak et al. (2015) targets by using a
# a function to customize the parameters
# (note that this yields exactly the same targets as p1, and simply
# offers an alternative for specifying targets with default parameters)
p3 <-
  p0 %>%
  add_auto_targets(method = spec_polak_targets(common_relative_target = 0.3))

# solve problems
s <- c(solve(p1), solve(p2), solve(p3))
names(s) <- c(
  "default Polak targets", "default Polak targets", "modified Polak targets"
)

# plot solutions
plot(s, axes = FALSE)

# in the previous examples, we specified the targets for each of the features
# based on the same target setting method. we can also specify a particular
# target setting method for each feature. to achieve this, we can
# provide the targets as a list. for example, here we will specify that
# the first 10 features should have their targets based on
# Jung et al. (2021) targets (with default parameters), the following 15
# features should have their targets based on Ward et al. (2025) targets
# (with default parameters), and all the remaining features should have
# their targets based on modified Jung et al. (2021) targets.

# note that add_group_targets provides a more convenient interface for
# specifying targets for multiple features

# create a list with the target setting methods
target_list <- append(
  append(
    rep(list(spec_jung_targets()), 10),
    rep(list(spec_ward_targets()), 15)
  ),
   rep(
    list(spec_jung_targets(prop_uplift = 0.3)),
    terra::nlyr(sim_complex_features) - 25
  )
```

```
)

# create problem with the list of target setting methods
p4 <- p0 %>% add_auto_targets(method = target_list)

# solve problem
s4 <- solve(p4)

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

# here we will show how to set the feature_units when the feature
# are not terra::rast() objects. in this example, we have planning units
# stored in an sf object (i.e., sim_pu_polygons) and the feature data will
# be stored as columns in the sf object.

# we will start by simulating feature data for the planning units.
# in particular, the simulated values will describe the amount of habitat
# for each feature expressed as acres (e.g., a value of 30 means 30 acres)
sim_pu_polygons$feature_1 <- runif(nrow(sim_pu_polygons), 0, 500)
sim_pu_polygons$feature_2 <- runif(nrow(sim_pu_polygons), 0, 600)
sim_pu_polygons$feature_3 <- runif(nrow(sim_pu_polygons), 0, 300)

# we will now build a problem with these data and specify the
# the feature units as acres because that those are the units we
# used for simulating the data. also, we will specify targets
# of 2 km^2 of habitat for each feature. although the feature units are
# acres, the function is able to able to automatically convert the units.
p5 <-
  problem(
    sim_pu_polygons,
    c("feature_1", "feature_2", "feature_3"),
    cost_column = "cost",
    feature_units = "acres"
  ) %>%
  add_min_set_objective() %>%
  add_auto_targets(
    method = spec_area_targets(targets = 2, area_units = "km^2")
  ) %>%
  add_binary_decisions() %>%
  add_default_solver(gap = 0, verbose = FALSE)

# solve problem
s5 <- solve(p5)

# plot solution
plot(s5[, "solution_1"], axes = FALSE)

## 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() 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). Only a single decision should be added to a problem() object. Note that if multiple decisions are added to an object, then the last one to be added will be used.

Value

An updated problem() object with the decisions added to it.

See Also

See decisions for an overview of all functions for adding decisions.

Other decisions: add_proportion_decisions(), add_semicontinuous_decisions()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_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(verbose = FALSE)
```

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

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

# create a matrix with targets for a multi-zone conservation problem
targs <- matrix(runif(15, 0.1, 0.2), nrow = 5, ncol = 3)

# build multi-zone conservation problem with binary decisions
p2 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  add_min_set_objective() %>%
  add_relative_targets(targs)  %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve the problem
s2 <- solve(p2)

# print solution
print(s2)

# plot solution
plot(category_layer(s2), main = "solution", axes = 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 (i.e., total perimeter).

Usage

```
## S4 method for signature 'ConservationProblem,ANY,ANY,ANY,ANY,data.frame'
add_boundary_penalties(x, penalty, edge_factor, formulation, zones, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,ANY,ANY,matrix'
add_boundary_penalties(x, penalty, edge_factor, formulation, zones, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,ANY,ANY,ANY'
add_boundary_penalties(x, penalty, edge_factor, formulation, zones, data)
```

Arguments

x	problem() 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.
formulation	character value denoting the name of the linearization technique used to formulate the penalties. Available options are "simple" and "knapsack". Defaults to "simple". If you are having issues with solving problems within a reasonable period of time using the "simple" technique, then consider trying the "knapsack" technique. Note that the "knapsack" technique is likely to yield better performance when each planning unit has a large number of neighbors (e.g., planning units are based on a hexagonal grid or irregular polygons).
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.

Value

An updated problem() object 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.

data as a NULL value indicating that the data should be 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).

data as a matrix/Matrix object 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 denote the total boundary length associated with each planning unit.

data as a data.frame object 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. Additionally, if the values in the "id1" and "id2" columns contain the same values, then the value denotes the amount of exposed boundary length (not total boundary). This format follows the the standard *Marxan* format for boundary data (i.e., per the "bound.dat" file).

Mathematical formulation

The boundary penalties are implemented 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.

$$\sum_{i}^{I}\sum_{z}^{Z}(p \times W_{zz}B_{ii}) + \sum_{i}^{I}\sum_{j}^{I}\sum_{z}^{Z}\sum_{y}^{Z}(-2 \times p \times X_{iz} \times X_{jy} \times W_{zy} \times B_{ij})$$

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 $-p$. Additionally, to linearize the problem, the expression $X_{iz} \times X_{jy}$ is modeled using a set of continuous variables (bounded between 0 and 1) based on Beyer *et al.* (2016).

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

See penalties for an overview of all functions for adding penalties. Also see add_neighbor_penalties() for a penalty that can reduce spatial fragmentation and has faster solver run times. Additionally, see calibrate_cohon_penalty() for assistance with selecting an appropriate penalty value.

Other functions for adding penalties: add_asym_connectivity_penalties(), add_connectivity_penalties(), add_feature_weights(), add_linear_penalties(), add_neighbor_penalties()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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)

# solve problems
s1 <- c(solve(p1), solve(p2), solve(p3), solve(p4))
names(s1) <- c("basic solution", "small penalties", "high penalties",
  "precomputed data"
)

# plot solutions
plot(s1, axes = FALSE)

# 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_zones_pu_raster, sim_zones_features) %>%
```

```
  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)

# solve problems
s2 <- list(solve(p5), solve(p6), solve(p7), solve(p8), solve(p9), solve(p10))

#convert to category layers for visualization
s2 <- terra::rast(lapply(s2, category_layer))
names(s2) <- c(
```

```
  "basic solution", "within zone clumping (low)",
  "within zone clumping (high)", "between zone clumping",
  "within + between clumping", "negative clumping"
)

# plot solutions
plot(s2, axes = FALSE)

## End(Not run)
```

---

add_cbc_solver *Add a* CBC *solver*

Description

Specify that the *CBC* (COIN-OR branch and cut) software should be used to solve a conservation planning problem (Forrest & Lougee-Heimer 2005). This function can also be used to customize the behavior of the solver. It requires the **rcbc** package to be installed (only available on GitHub, see below for installation instructions).

Usage

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

Arguments

x	problem() object.
gap	numeric gap to optimality. This gap is relative and expresses the acceptable deviance from the optimal objective. For example, a value of 0.01 will result in the solver stopping when it has found a solution within 1% of optimality. Additionally, a value of 0 will result in the solver stopping when it has found an optimal solution. The default value is 0.1 (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: (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.

`start_solution` NULL or object containing the starting solution for the solver. This is can be useful because specifying a starting solution can speed up the optimization process. 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, `terra::rast()`, 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.

`control` list with additional parameters for tuning the optimization process. For example, `control = list(strategy = 2)` could be used to set the `strategy` parameter. See the online documentation for information on the parameters.

Details

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

Value

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

Installation

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

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

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

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** `terra::rast()` **planning units** The argument to `start_solution` be a `terra::rast()` object where different cells 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** `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

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

See Also

Other functions for adding solvers: `add_cplex_solver()`, `add_default_solver()`, `add_gurobi_solver()`, `add_highs_solver()`, `add_lsymphony_solver`, `add_rsymphony_solver()`

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()

# create problem
p1 <-
  problem(sim_pu_raster, sim_features) %>%
  add_min_set_objective() %>%
  add_relative_targets(0.1) %>%
  add_binary_decisions() %>%
  add_cbc_solver(gap = 0, verbose = FALSE)

# generate solution %>%
s1 <- solve(p1)

# plot solution
plot(s1, main = "solution", axes = 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_cbc_solver(gap = 0, start_solution = s1, verbose = FALSE)

# generate solution
s2 <- solve(p2)

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

## End(Not run)
```

---

add_connectivity_penalties

Add connectivity penalties

Description

Add penalties to a conservation planning problem to account for symmetric connectivity between planning units. Symmetric connectivity data describe connectivity information that is not directional. For example, symmetric connectivity data could describe which planning units are adjacent to each other (see `adjacency_matrix()`), or which planning units are within threshold distance of each other (see `proximity_matrix()`).

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() 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 working with multiple zones and the argument to data is a matrix or Matrix object. If the argument to data is an array or data.frame with data for multiple zones (e.g., using the "zone1" and "zone2" column names), 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 (based on Önal and Briers 2002).

Value

An updated `problem()` object with the penalties added to it.

Data format

The argument to `data` can be specified using several different formats.

`data` **as a** `matrix`**/**`Matrix` **object** 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` **as a** `data.frame` **object** containing columns that are named `"id1"`, `"id2"`, and `"boundary"`. Here, each row denotes the connectivity between a pair of planning units (per values in the `"id1"` and `"id2"` columns) following the *Marxan* format. If the argument to `x` contains multiple zones, then the `"zone1"` and `"zone2"` columns can optionally be provided to manually specify the connectivity values between planning units when they are allocated to specific zones. If the `"zone1"` and `"zone2"` columns are present, then the argument to `zones` must be `NULL`.

`data` **as an** `array` **object** 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.

Mathematical formulation

The connectivity penalties are implemented 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. Additionally, to linearize the problem, the expression $X_{iz} \times X_{jy}$ is modeled using a set of continuous variables (bounded between 0 and 1) based on Beyer *et al.* (2016).

Notes

In previous versions, this function aimed to handle both symmetric and asymmetric connectivity data. This meant that the mathematical formulation used to account for asymmetric connectivity was different to that implemented by the *Marxan* software (see Beger *et al.* for details). To ensure that asymmetric connectivity is handled in a similar manner to the *Marxan* software, the `add_asym_connectivity_penalties()` function should now be used for asymmetric connectivity data.

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.

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

Önal H, and Briers RA (2002) Incorporating spatial criteria in optimum reserve network selection. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 269: 2437–2441.

See Also

See penalties for an overview of all functions for adding penalties. Also see `add_asym_connectivity_penalties()` to account for asymmetric connectivity between planning units. Additionally, see `calibrate_cohon_penalty()` for assistance with selecting an appropriate `penalty` value.

Other functions for adding penalties: `add_asym_connectivity_penalties()`, `add_boundary_penalties()`, `add_feature_weights()`, `add_linear_penalties()`, `add_neighbor_penalties()`

Examples

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

# load data
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()
```

```
# 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)

# rescale matrix values to have a maximum value of 1
b_matrix <- rescale_matrix(b_matrix, max = 1)

# visualize connectivity matrix
Matrix::image(b_matrix)

# 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 <- sf::st_coordinates(
  suppressWarnings(sf::st_centroid(sim_pu_polygons))
)
d_matrix <- (1 / (Matrix::Matrix(as.matrix(dist(centroids))) + 1))

# rescale matrix values to have a maximum value of 1
d_matrix <- rescale_matrix(d_matrix, max = 1)

# remove connections between planning units with values below a threshold to
# reduce run-time
d_matrix[d_matrix < 0.8] <- 0

# visualize connectivity matrix
Matrix::image(d_matrix)

# create a symmetric connectivity matrix where the connectivity
# between adjacent two planning units corresponds to their combined
# value in a column of the planning unit data

# for example, this column 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")

# rescale matrix values to have a maximum value of 1
c_matrix <- rescale_matrix(c_matrix, max = 1)

# visualize connectivity matrix
Matrix::image(c_matrix)
```

```
# 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)
)

# solve problems
s2 <- lapply(p2, solve)

# create single object with all solutions
s2 <- sf::st_sf(
  tibble::tibble(
    p2_1 = s2[[1]]$solution_1,
    p2_2 = s2[[2]]$solution_1,
    p2_3 = s2[[3]]$solution_1,
    p2_4 = s2[[4]]$solution_1,
    p2_5 = s2[[5]]$solution_1,
    p2_6 = s2[[6]]$solution_1,
    p2_7 = s2[[7]]$solution_1
  ),
  geometry = sf::st_geometry(s2[[1]])
)

names(s2)[1:7] <- c(
  "basic problem",
  paste0("b_matrix (", penalties,")"),
  paste0("d_matrix (", penalties,")"),
  paste0("c_matrix (", penalties,")")
)

# plot solutions
plot(s2)

# create minimal multi-zone problem and limit solver to one minute
# to obtain solutions in a short period of time
p3 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  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_zones_pu_raster)
```

```
# visualize matrix
Matrix::image(a_matrix)

# 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)
)

# solve problems
s4 <- lapply(p4, solve)
s4 <- lapply(s4, category_layer)
s4 <- terra::rast(s4)
names(s4) <-  c(
  "basic problem",
  paste0("zm", rep(seq_len(5), each = 2), " (", rep(penalties2, 2), ")")
)

# plot solutions
plot(s4, axes = FALSE)

# 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_zones_pu_raster[[1]]), 2), 3, 3))
for (z1 in seq_len(3))
  for (z2 in seq_len(3))
    c_array[, , z1, z2] <- round(
      runif(ncell(sim_zones_pu_raster[[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)
)
```

```
# solve problems
s5 <- lapply(p5, solve)
s5 <- lapply(s5, category_layer)
s5 <- terra::rast(s5)
names(s5) <- c("basic problem", "connectivity array")

# plot solutions
plot(s5, axes = 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()` 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

An updated problem() object with the constraints added to it.

Data format

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

data **as a** NULL **value** indicating that 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).

data **as a** matrix**/**Matrix **object** 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 **as a** data.frame **object** containing columns that are named "id1", "id2", and "boundary". Here, each row denotes the connectivity between two planning units following the *Marxan* format. The "boundary" column should contain binary numeric values that indicate if the two planning units specified in the "id1" and "id2" columns 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

See constraints for an overview of all functions for adding constraints.

Other functions for adding constraints: add_feature_contiguity_constraints(), add_linear_constraints(), add_locked_in_constraints(), add_locked_out_constraints(), add_mandatory_allocation_constraints(), add_manual_bounded_constraints(), add_manual_locked_constraints(), add_neighbor_constraints()

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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()

# solve problems
s1 <- c(solve(p1), solve(p2))
names(s1) <- c("basic solution", "connected solution")

# plot solutions
plot(s1, axes = FALSE)

# 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_zones_pu_raster, sim_zones_features) %>%
  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 3 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)

# solve problems
s2 <- lapply(list(p3, p4, p5, p6), solve)
s2 <- lapply(s2, category_layer)
s2 <- terra::rast(s2)
names(s2) <- c("basic solution", "p4", "p5", "p6")

# plot solutions
plot(s2, axes = FALSE)

# 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 50% 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 30% 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_zones_pu_raster[[c(1, 1)]]
pus[[2]] <- pus[[2]] * 0.3
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 = terra::global(sim_features, "sum", na.rm = TRUE)[[1]] * 0.5,
  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)

# solve problems
s7 <- category_layer(solve(p7))

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

## End(Not run)
```

`add_cplex_solver` *Add a* CPLEX *solver*

Description

Specify that the *IBM CPLEX* software should be used to solve a conservation planning problem (IBM 2017) . 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()` 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/cplex-optimizer>).

Value

An updated problem() object with the solver added to it.

Installation

The **cplexAPI** package is used to interface with *IBM CPLEX* software. To install the package, the *IBM CPLEX* software must be installed (see https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-optimizer). Next, the CPLEX_BIN environmental variable must be set to specify the file path for the *IBM 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"
```

Please Note that you may need to change the version number in the file path (i.e., "CPLEX_Studio128"). After specifying the CPLEX_BIN environmental variable, the **cplexAPI** package can be installed. Since the **cplexAPI** package is not available on the the Comprehensive R Archive Network (CRAN), it must be installed from its GitHub repository. To install the **cplexAPI** package, please use the following code:

```
if (!require(remotes)) install.packages("remotes")
remotes::install_github("cran/cplexAPI")
```

For further details on installing this package, please consult the installation instructions.

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

Other functions for adding solvers: `add_cbc_solver()`, `add_default_solver()`, `add_gurobi_solver()`, `add_highs_solver()`, `add_lsymphony_solver`, `add_rsymphony_solver()`

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_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)

## 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 = 10)
```

Arguments

`x`	`problem()` 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

An updated `problem()` object 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

See portfolios for an overview of all functions for adding a portfolio.

Other functions for adding portfolios: `add_default_portfolio()`, `add_extra_portfolio()`, `add_gap_portfolio()`, `add_shuffle_portfolio()`, `add_top_portfolio()`

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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)

# solve problem and generate 10 solutions within 20% of optimality
s1 <- solve(p1)
```

```
# convert portfolio into a multi-layer raster object
s1 <- terra::rast(s1)

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

# build multi-zone conservation problem with cuts portfolio
p2 <-
 problem(sim_zones_pu_raster, sim_zones_features) %>%
 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)

# solve the problem
s2 <- solve(p2)

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

# convert each solution in the portfolio into a single category layer
s2 <- terra::rast(lapply(s2, category_layer))

# plot solutions in portfolio
plot(s2, main = "solution", axes = FALSE)

## End(Not run)
```

`add_default_portfolio` *Add a default portfolio*

Description

Generate a portfolio containing a single solution.

Usage

```
add_default_portfolio(x)
```

Arguments

x	`problem()` object.

Details

By default, this is portfolio is added to `problem()` objects if no other portfolios is manually specified.

Value

An updated problem() object with the portfolio added to it.

See Also

See portfolios for an overview of all functions for adding a portfolio.

Other functions for adding portfolios: add_cuts_portfolio(), add_extra_portfolio(), add_gap_portfolio(), add_shuffle_portfolio(), add_top_portfolio()

---

add_default_solver *Add default solver*

Description

Specify that the best solver currently available should be used to solve a conservation planning problem.

Usage

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

Arguments

x	problem() 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_highs_solver(), add_lpsymphony_solver(), and finally add_rsymphony_solver(). For information on the performance of different solvers, please see Schuster *et al.* (2020).

Value

An updated problem() object 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

See solvers for an overview of all functions for adding a solver.

Other functions for adding solvers: add_cbc_solver(), add_cplex_solver(), add_gurobi_solver(), add_highs_solver(), add_lsymphony_solver, add_rsymphony_solver()

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() 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

An updated problem() object with the portfolio added to it.

See Also

See portfolios for an overview of all functions for adding a portfolio.

Other functions for adding portfolios: add_cuts_portfolio(), add_default_portfolio(), add_gap_portfolio(), add_shuffle_portfolio(), add_top_portfolio()

Examples

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

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_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)

# convert portfolio into a multi-layer raster object
s1 <- terra::rast(s1)

# print number of solutions found
print(terra::nlyr(s1))

# plot solutions
plot(s1, axes = FALSE)

# create multi-zone problem with a portfolio for extra solutions
p2 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  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)

# convert each solution in the portfolio into a single category layer
s2 <- terra::rast(lapply(s2, category_layer))

# print number of solutions found
print(terra::nlyr(s2))

# plot solutions in portfolio
plot(s2, axes = 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 throughout the areas designated for their conservation.

Usage

```
## 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() 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

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

Data format

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

`data` **as a** `NULL` **value** 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).

`data` **as a** `matrix`/`Matrix` **object** 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` **as a** `data.frame` **object** containing columns that are named `"id1"`, `"id2"`, and `"boundary"`. Here, each row denotes the connectivity between two planning units following the *Marxan* format. The `"boundary"` column should contain binary `numeric` values that indicate if the two planning units specified in the `"id1"` and `"id2"` columns 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.

`data` **as a** `list` **object** 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

See constraints for an overview of all functions for adding constraints.

Other functions for adding constraints: add_contiguity_constraints(), add_linear_constraints(), add_locked_in_constraints(), add_locked_out_constraints(), add_mandatory_allocation_constraints(), add_manual_bounded_constraints(), add_manual_locked_constraints(), add_neighbor_constraints()

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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 5th percentile)
cm4 <- lapply(seq_len(terra::nlyr(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 0/1 values denoting values in top 5th percentile
  m <- round(m > quantile(as.vector(m), 1 - 0.05, names = FALSE))
  # remove 0s from the sparse matrix
  m <- Matrix::drop0(m)
  # return matrix
  m
})
p4 <- p1 %>% add_feature_contiguity_constraints(data = cm4)

# solve problems
s1 <- c(solve(p1), solve(p2), solve(p3), solve(p4))
names(s1) <- c(
```

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

# 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_zones_pu_raster, sim_zones_features) %>%
  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_zones_features)),
  function(i) matrix(ifelse(i == 1, 1, 0), ncol = 3, nrow = 3)
)
print(zm8)
p8 <- p5 %>% add_feature_contiguity_constraints(zm8)

# solve problems
s2 <- lapply(list(p5, p6, p7, p8), solve)
s2 <- terra::rast(lapply(s2, category_layer))
names(s2) <- c("p5", "p6", "p7", "p8")
# plot solutions
plot(s2, axes = FALSE)

## End(Not run)
```

`add_feature_weights` *Add feature weights*

Description

Add features weights to a conservation planning problem. Specifically, some objective functions aim to maximize (or minimize) a metric that measures how well a set of features are represented by a solution (e.g., maximize the number of features that are adequately represented, `add_max_features_objective()`). In such cases, it may be desirable to prefer the representation of some features over other features (e.g., features that have higher extinction risk might be considered more important than those with lower extinction risk). To achieve this, weights can be used to specify how much more important it is for a solution to represent particular features compared with other features.

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()` 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()`, `add_min_shortfall_objective()`). They can also 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

An updated `problem()` with the weights added to it.

Weights format

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

`weights` **as a** `numeric` **vector** containing weights for each feature. Note that this format cannot be used to specify weights for problems with multiple zones.

`weights` **as a** `matrix` **object** 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

See penalties for an overview of all functions for adding penalties.

Other functions for adding penalties: `add_asym_connectivity_penalties()`, `add_boundary_penalties()`, `add_connectivity_penalties()`, `add_linear_penalties()`, `add_neighbor_penalties()`

Examples

```
## Not run:
# load package
require(ape)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_phylogeny <- get_sim_phylogeny()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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 / terra::global(sim_features, "sum", na.rm = TRUE)[[1]]) * 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)

# solve problems
s1 <- c(solve(p1), solve(p2), solve(p3))
```

```
names(s1) <- c("equal weights", "rarity weights", "manual weights")

# plot solutions
plot(s1, axes = FALSE)

# plot the example phylogeny
par(mfrow = c(1, 1))
plot(sim_phylogeny, main = "simulated phylogeny")

# 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)

# solve problem
s4 <- solve(p4)

# plot solution
plot(s4, main = "solution", axes = 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", terra::nlyr(sim_features)), which(r4$met), "red"
  )
)

# 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, 10000, 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)
```

```
# 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)

# 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", terra::nlyr(sim_features)), which(r5$met), "red"
  )
)

# 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_zones_pu_raster, sim_zones_features) %>%
  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)

# solve problems
s6 <- solve(p6)
s7 <- solve(p7)

# convert solutions to category layers
c6 <- category_layer(s6)
c7 <- category_layer(s7)

# plot solutions
```

```
plot(c(c6, c7), main = c("equal weights", "manual weights"), axes = FALSE)

# 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("feature_1", "feature_4"),
    type = "relative",
    target = 0.1)
  ) %>%
  add_feature_weights(matrix(c(1, 200), ncol = 1)) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s8 <- solve(p8)

# plot solution
plot(s8, main = "solution", axes = 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 = 10, pool_gap = 0.1)
```

Arguments

`x`	`problem()` object.
`number_solutions`	`integer` number of solutions required. Defaults to 10.
`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

An updated `problem()` object with the portfolio added to it.

See Also

See portfolios for an overview of all functions for adding a portfolio.

Other functions for adding portfolios: `add_cuts_portfolio()`, `add_default_portfolio()`, `add_extra_portfolio()`, `add_shuffle_portfolio()`, `add_top_portfolio()`

Examples

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

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_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)

# convert portfolio into a multi-layer raster
s1 <- terra::rast(s1)

# print number of solutions found
print(terra::nlyr(s1))

# plot solutions
plot(s1, axes = FALSE)
```

```
# create multi-zone  problem with a portfolio containing 10 solutions within
# 20% of optimality
p2 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  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)

# convert portfolio into a multi-layer raster of category layers
s2 <- terra::rast(lapply(s2, category_layer))

# print number of solutions found
print(terra::nlyr(s2))

# plot solutions in portfolio
plot(s2, axes = FALSE)

## End(Not run)
```

`add_group_targets` *Add targets based on feature groups*

Description

Add targets to a conservation planning problem, wherein each feature is assigned to a particular group and a target setting method is specified for each feature group. This function is designed to provide a convenient alternative to `add_auto_targets()`.

Usage

```
add_group_targets(x, groups, method)
```

Arguments

`x`	`problem()` object.
`groups`	`character` vector of group names.
`method`	`list` of name-value pairs that describe the target setting method for each group. Here, the names of `method` correspond to different values in `groups`. Additionally, the elements of `method` should be the `character` names of target setting methods, or objects (`TargetMethod`) that specify target setting methods. See Target methods below for more information.

Value

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

Target methods

Here `method` is used to specify a target setting method for each group. In particular, each element should correspond to a different group, with the name of the element indicating the group and the value denoting the target setting method. One option for specifying a particular target setting method is to use a `character` value that denotes the name of the method (e.g., `"jung"` to set targets following Jung *et al.* 2021). This option is particularly useful if default parameters should be considered during target calculations. Alternatively, another option for specifying a particular target setting method is to use a function to define an object (e.g., `spec_jung_targets()` to set targets following Jung *et al.* 2021). This alternative option is particularly useful if customized parameters should be considered during target calculations. Note that a `method` can contain a mix of target setting methods defined using `character` values and functions.

The following `character` values can be used to specify target setting methods: `"jung"` (per Jung *et al.* 2021), `"polak"` (per Polak *et al.* 2016), `"rodrigues"` (per Rodrigues *et al.* 2004), `"ward"` (per Ward *et al.* 2025), and `"watson"` (per Watson *et al.* 2010). (2010). Additionally, the following values can be used to set targets based on criteria from the IUCN Red List of Threatened Species (IUCN 2025): `"rl_species_VU_A1_B1"` `"rl_species_EN_A1_B1"`, `"rl_species_CR_A1_B1"`, `"rl_species_VU_A1_B2"` `"rl_species_EN_A1_B2"`, `"rl_species_CR_A1_B2"`, `"rl_species_VU_A2_B1"` `"rl_species_EN_A2_B1"`, `"rl_species_CR_A2_B1"`, `"rl_species_VU_A2_B2"` `"rl_species_EN_A2_B2"`, `"rl_species_CR_A2_B2"`, `"rl_species_VU_A3_B1"` `"rl_species_EN_A3_B1"`, `"rl_species_CR_A3_B1"`, `"rl_species_VU_A3_B2"` `"rl_species_EN_A3_B2"`, `"rl_species_CR_A3_B2"`, `"rl_species_VU_A4_B1"` `"rl_species_EN_A4_B1"`, `"rl_species_CR_A4_B1"`, `"rl_species_VU_A4_B2"` `"rl_species_EN_A4_B2"`, and `"rl_species_CR_A4_B2"`. Furthermore, the following values can be used to set targets based on criteria from the IUCN Red List of Ecosystems (IUCN 2024): `"rl_ecosystem_VU_A1_B1"`, `"rl_ecosystem_EN_A1_B1"`, `"rl_ecosystem_CR_A1_B1"`, `"rl_ecosystem_VU_A1_B2"`, `"rl_ecosystem_EN_A1_B2"`, `"rl_ecosystem_CR_A1_B2"`, `"rl_ecosystem_VU_a2a_B1"`, `"rl_ecosystem_EN_a2a_B1"`, `"rl_ecosystem_CR_a2a_B1"`, `"rl_ecosystem_VU_a2a_B2"`, `"rl_ecosystem_EN_a2a_B2"`, `"rl_ecosystem_CR_a2a_B2"`, `"rl_ecosystem_VU_a2b_B1"`, `"rl_ecosystem_EN_a2b_B1"`, `"rl_ecosystem_CR_a2b_B1"`, `"rl_ecosystem_VU_a2b_B2"`, `"rl_ecosystem_EN_a2b_B2"`, `"rl_ecosystem_CR_a2b_B2"`, `"rl_ecosystem_VU_A3_B1"`, `"rl_ecosystem_EN_A3_B1"`, `"rl_ecosystem_CR_A3_B1"`, `"rl_ecosystem_VU_A3_B2"`, `"rl_ecosystem_EN_A3_B2"`, and `"rl_ecosystem_CR_A3_B2"`.' For convenience, these options can also be specified with lower case letters (e.g., `"rl_species_VU_A1_B1"` may alternatively be specified as `"rl_species_vu_a1_b1"`). Note that target setting methods that require additional data or parameters cannot be specified with `character` values.

The following functions can be used to create an object for specifying a target setting method: `spec_absolute_targets()`, `spec_relative_targets()`, `spec_area_targets()`, `spec_interp_absolute_targets()`, `spec_interp_area_targets()`, `spec_duran_targets()`, `spec_jung_targets()`, `spec_polak_targets()`, `spec_pop_size_targets()`, `spec_rl_ecosystem_targets()`, `spec_rl_species_targets()`, `spec_rodrigues_targets()`, `spec_rule_targets()`, `spec_ward_targets()`, `spec_watson_targets()`, `spec_wilson_targets()`, `spec_min_targets()`, and `spec_max_targets()`.

Target setting

Many conservation planning problems require targets. Targets are used to specify the minimum amount, or proportion, of a feature's spatial distribution that should ideally be protected. This is important so that the optimization process can weigh the merits and trade-offs between improving the representation of one feature over another feature. Although it can be challenging to set meaningful targets, this is a critical step for ensuring that prioritizations meet the stakeholder objectives that underpin a prioritization exercise (Carwardine *et al.* 2009). In other words, targets play an important role in ensuring that a priority setting process is properly tuned according to stakeholder

requirements. For example, targets provide a mechanism for ensuring that a prioritization secures enough habitat to promote the long-term persistence of each threatened species, culturally important species, or economically important ecosystem services under consideration. Since there is often uncertainty regarding stakeholder objectives (e.g., how much habitat should be protected for a given species) or the influence of particular target on a prioritization (e.g., how would setting a 90% or 100% for a threatened species alter priorities), it is often useful to generate and compare a suite of prioritizations based on different target scenarios.

Data calculations

Many of the functions for specifying target setting methods involve calculating targets based on the spatial extent of the features in `x` (e.g., `spec_jung_targets()`, [spec_rodrigues_targets(), and others). Although this function for adding targets can be readily applied to `problem()` objects that have the feature data provided as a `terra::rast()` object, you will need to specify the spatial units for the features when building a `problem()` object if the feature data are provided in a different format. In particular, if the feature data are provided as a `data.frame` or `character` vector, then you will need to specify an argument to `feature_units` when using the `problem()` function. See the Examples section below for a demonstration of using the `feature_units` parameter.

References

Carwardine J, Klein CJ, Wilson KA, Pressey RL, Possingham HP (2009) Hitting the target and missing the point: target-based conservation planning in context. *Conservation Letters*, 2: 4–11.

Jung M, Arnell A, de Lamo X, García-Rangel S, Lewis M, Mark J, Merow C, Miles L, Ondo I, Pironon S, Ravilious C, Rivers M, Schepaschenko D, Tallowin O, van Soesbergen A, Govaerts R, Boyle BL, Enquist BJ, Feng X, Gallagher R, Maitner B, Meiri S, Mulligan M, Ofer G, Roll U, Hanson JO, Jetz W, Di Marco M, McGowan J, Rinnan DS, Sachs JD, Lesiv M, Adams VM, Andrew SC, Burger JR, Hannah L, Marquet PA, McCarthy JK, Morueta-Holme N, Newman EA, Park DS, Roehrdanz PR, Svenning J-C, Violle C, Wieringa JJ, Wynne G, Fritz S, Strassburg BBN, Obersteiner M, Kapos V, Burgess N, Schmidt- Traub G, Visconti P (2021) Areas of global importance for conserving terrestrial biodiversity, carbon and water. *Nature Ecology and Evolution*, 5:1499–1509.

Polak T, Watson JEM, Fuller RA, Joseph LN, Martin TG, Possingham HP, Venter O, Carwardine J (2015) Efficient expansion of global protected areas requires simultaneous planning for species and ecosystems. *Royal Society Open Science*, 2: 150107.

See Also

Other functions for adding targets: `add_absolute_targets()`, `add_auto_targets()`, `add_manual_targets()`, `add_relative_targets()`

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_complex_pu_raster <- get_sim_complex_pu_raster()
sim_complex_features <- get_sim_complex_features()
```

```
sim_pu_polygons <- get_sim_pu_polygons()

# create grouping data, wherein each feature will be randomly
# assigned to the group A, B, C, or D
sim_groups <- sample(
  c("A", "B", "C", "D"), terra::nlyr(sim_complex_features), replace = TRUE
)

# create problem where each feature in group A is assigned a 20% target,
# each feature in group B is assigned a target based on Jung et al. (2021),
# each feature in group C is assigned a target based on Ward et al. (2025),
# and each feature in group D is assigned a target based on Rodrigues
# et al. (2004)
p1 <-
  problem(sim_complex_pu_raster, sim_complex_features) %>%
  add_min_set_objective() %>%
  add_group_targets(
    groups = sim_groups,
    method = list(
      A = spec_relative_targets(0.2),
      B = spec_jung_targets(),
      C = spec_ward_targets(),
      D = spec_rodrigues_targets()
    )
  ) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s1 <- solve(p1)

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

# here we will show how to set the feature_units when the feature
# are not terra::rast() objects. in this example, we have planning units
# stored in an sf object (i.e., sim_pu_polygons) and the feature data will
# be stored as columns in the sf object.

# we will start by simulating feature data for the planning units.
# in particular, the simulated values will describe the amount of habitat
# for each feature expressed as acres (e.g., a value of 30 means 30 acres).
sim_pu_polygons$feature_1 <- runif(nrow(sim_pu_polygons), 0, 500)
sim_pu_polygons$feature_2 <- runif(nrow(sim_pu_polygons), 0, 600)
sim_pu_polygons$feature_3 <- runif(nrow(sim_pu_polygons), 0, 300)

# we will now build a problem with these data and specify the
# the feature units as acres because that those are the units we
# used for simulating the data. also, we will specify targets
# of 2 km^2 of habitat for the first feature, and 3 km^2 for the
# remaining two features by assigning the features to groups.
# although the feature units are acres, the function is able to able
# to automatically convert the units.
```

```
p2 <-
  problem(
    sim_pu_polygons,
    c("feature_1", "feature_2", "feature_3"),
    cost_column = "cost",
    feature_units = "acres"
  ) %>%
  add_min_set_objective() %>%
  add_group_targets(
    groups = c("A", "B", "B"),
    method = list(
      A = spec_area_targets(targets = 2, area_units = "km^2"),
      B = spec_area_targets(targets = 3, area_units = "km^2")
    )
  ) %>%
  add_binary_decisions() %>%
  add_default_solver(gap = 0, verbose = FALSE)

# solve problem
s2 <- solve(p2)

# plot solution
plot(s2[, "solution_1"], axes = FALSE)

## End(Not run)
```

`add_gurobi_solver` *Add a* Gurobi *solver*

Description

Specify that the *Gurobi* software should be used to solve a conservation planning problem (Gurobi Optimization LLC 2021). 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,
  node_file_start = Inf,
  start_solution = NULL,
  verbose = TRUE,
  control = list()
)
```

Arguments

x	problem() object.
gap	numeric gap to optimality. This gap is relative and expresses the acceptable deviance from the optimal objective. For example, a value of 0.01 will result in the solver stopping when it has found a solution within 1% of optimality. Additionally, a value of 0 will result in the solver stopping when it has found an optimal solution. The default value is 0.1 (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 with problems that may suffer from numerical instability issues. Beware that it will likely substantially increase run time (sets the *Gurobi* NumericFocus parameter to 2). Defaults to FALSE.
node_file_start	numeric threshold amount of memory (in GB). Once the amount of memory (RAM) used to store information for solving the optimization problem exceeds this parameter value, the solver will begin storing this information on disk (using the *Gurobi* NodeFileStart parameter). This functionality is useful if the system has insufficient memory to solve a given problem (e.g., solving the problem with default settings yields the OUT OF MEMORY error message) and a system with more memory is not readily available. For example, a value of 4 indicates that the solver will start using the disk after it uses more than 4 GB of memory to store information on solving the problem. Defaults to Inf such that the solver will not attempt to store information on disk when solving a given problem.
start_solution	NULL or object containing the starting solution for the solver. This is can be useful because specifying a starting solution can speed up the optimization process. 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, terra::rast(), 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`.

control `list` with additional parameters for tuning the optimization process. For example, `control = list(Method = 2)` could be used to set the `Method` parameter. See the online documentation for information on the parameters.

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

An updated `problem()` object 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_guide", 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** `terra::rast()` **planning units** The argument to `start_solution` be a `terra::rast()` object where different cells 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** `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

See solvers for an overview of all functions for adding a solver.

Other functions for adding solvers: `add_cbc_solver()`, `add_cplex_solver()`, `add_default_solver()`, `add_highs_solver()`, `add_lsymphony_solver`, `add_rsymphony_solver()`

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()

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

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

# generate solution
s2 <- solve(p2)

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

## End(Not run)
```

`add_highs_solver` *Add a* HiGHS *solver*

Description

Specify that the *HiGHS* software should be used to solve a conservation planning problem (Huangfu and Hall 2018). This function can also be used to customize the behavior of the solver. It requires the **highs** package to be installed.

Usage

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

Arguments

x	problem() object.
gap	numeric gap to optimality. This gap is relative and expresses the acceptable deviance from the optimal objective. For example, a value of 0.01 will result in the solver stopping when it has found a solution within 1% of optimality. Additionally, a value of 0 will result in the solver stopping when it has found an optimal solution. The default value is 0.1 (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.
control	list with additional parameters for tuning the optimization process. For example, control = list(simplex_strategy = 1) could be used to set the simplex_strategy parameter. See the online documentation for information on the parameters.

Details

HiGHS is an open source optimization software. Although this solver can have comparable performance to the *CBC* solver (i.e., add_cbc_solver()) for particular problems and is generally faster than the *SYMPHONY* based solvers (i.e., add_rsymphony_solver(), add_lpsymphony_solver()), it can sometimes take much longer than the *CBC* solver for particular problems. This solver is recommended if the add_gurobi_solver(), add_cplex_solver(), add_cbc_solver() cannot be used.

Value

An updated problem() object with the solver added to it.

References

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

See Also

Other functions for adding solvers: add_cbc_solver(), add_cplex_solver(), add_default_solver(), add_gurobi_solver(), add_lsymphony_solver, add_rsymphony_solver()

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()

# create problem
p <-
  problem(sim_pu_raster, sim_features) %>%
  add_min_set_objective() %>%
  add_relative_targets(0.1) %>%
  add_binary_decisions() %>%
```

```
  add_highs_solver(gap = 0, verbose = FALSE)

# generate solution
s <- solve(p)

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

## End(Not run)
```

`add_linear_constraints` *Add linear constraints*

Description

Add constraints to a conservation planning problem to ensure that all selected planning units meet certain criteria.

Usage

```
## S4 method for signature 'ConservationProblem,ANY,ANY,character'
add_linear_constraints(x, threshold, sense, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,numeric'
add_linear_constraints(x, threshold, sense, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,matrix'
add_linear_constraints(x, threshold, sense, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,Matrix'
add_linear_constraints(x, threshold, sense, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,Raster'
add_linear_constraints(x, threshold, sense, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,SpatRaster'
add_linear_constraints(x, threshold, sense, data)

## S4 method for signature 'ConservationProblem,ANY,ANY,dgCMatrix'
add_linear_constraints(x, threshold, sense, data)
```

Arguments

`x`	`problem()` object.
`threshold`	`numeric` value. This threshold value is also known as a "right-hand-side" value per integer programming terminology.

sense `character` sense for the constraint. Available options include `">="`, `"<="`, or `"="` values.

data `character`, `numeric`, `terra::rast()`, `matrix`, or `Matrix` object containing the constraint values. These constraint values are also known as constraint coefficients per integer programming terminology. See the Data format section for more information.

Details

This function adds general purpose constraints that can be used to ensure that solutions meet certain criteria (see Examples section below for details). For example, these constraints can be used to add multiple budgets. They can also be used to ensure that the total number of planning units allocated to a certain administrative area (e.g., country) does not exceed a certain threshold (e.g., 30% of its total area). Furthermore, they can also be used to ensure that features have a minimal level of representation (e.g., 30%) when using an objective function that aims to enhance feature representation given a budget (e.g., `add_min_shortfall_objective()`).

Value

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

Mathematical formulation

The linear constraints are implemented using the following equation. 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 indicating if each planning unit is allocated or not). Also, let D_{iz} denote the constraint data associated with planning units $i \in I$ for zones $z \in Z$ (argument to `data`, if supplied as a `matrix` object), θ denote the constraint sense (argument to `sense`, e.g., $<=$), and t denote the constraint threshold (argument to `threshold`).

$$\sum_{i}^{I} \sum_{z}^{Z} (D_{iz} \times X_{iz}) \theta t$$

Data format

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

`data` **as** `character` **vector** containing column name(s) that contain penalty values for planning units. This format is only compatible if the planning units in the argument to `x` are a `sf::sf()` or `data.frame` object. The column(s) must have `numeric` values, and must not contain any missing (`NA`) values. For problems that contain a single zone, the argument to `data` must contain a single column name. Otherwise, for problems that contain multiple zones, the argument to `data` must contain a column name for each zone.

`data` **as a** `numeric` **vector** containing values for planning units. These values must not contain any missing (`NA`) values. Note that this format is only available for planning units that contain a single zone.

`data` **as a** `matrix`/`Matrix` **object** containing `numeric` values that specify data for 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.

data **as a** terra::rast() **object** containing values for planning units. This format is only compatible if the planning units in the argument to x are sf::sf(), or terra::rast() objects. If the planning unit data are a sf::sf() object, then the values are calculated by overlaying the planning units with the argument to data and calculating the sum of the values associated with each planning unit. If the planning unit data are a terra::rast() object, then the values are calculated by extracting the cell values (note that the planning unit data and the argument to data 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.

See Also

See constraints for an overview of all functions for adding constraints.

Other functions for adding constraints: add_contiguity_constraints(), add_feature_contiguity_constraints(), add_locked_in_constraints(), add_locked_out_constraints(), add_mandatory_allocation_constraints(), add_manual_bounded_constraints(), add_manual_locked_constraints(), add_neighbor_constraints()

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()

# create a baseline problem with minimum shortfall objective
p0 <-
  problem(sim_pu_raster, sim_features) %>%
  add_min_shortfall_objective(1800) %>%
  add_relative_targets(0.2) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s0 <- solve(p0)

# plot solution
plot(s0, main = "solution", axes = FALSE)

# now let's create some modified versions of this baseline problem by
# adding additional criteria using linear constraints

# first, let's create a modified version of p0 that contains
# an additional budget for a secondary cost dataset

# create a secondary cost dataset by simulating values
sim_pu_raster2 <- simulate_cost(sim_pu_raster)

# plot the primary cost dataset (sim_pu_raster) and
# the secondary cost dataset (sim_pu_raster2)
plot(
  c(sim_pu_raster, sim_pu_raster2),
  main = c("sim_pu_raster", "sim_pu_raster2"),
```

```
  axes = FALSE
)

# create a modified version of p0 with linear constraints that
# specify that the planning units in the solution must not have
# values in sim_pu_raster2 that sum to a total greater than 500
p1 <-
  p0 %>%
  add_linear_constraints(
    threshold = 500, sense = "<=", data = sim_pu_raster2
  )

# solve problem
s1 <- solve(p1)

# plot solutions s1 and s2 to compare them
plot(c(s0, s1), main = c("s0", "s1"), axes = FALSE)

# second, let's create a modified version of p0 that contains
# additional constraints to ensure that each feature definitely has
# at least 8% of its overall distribution represented by the solution
# (in addition to the 20% targets which specify how much we would
# ideally want to conserve for each feature)

# to achieve this, we need to calculate the total amount of each feature
# within the planning units so we can, in turn, set the constraint thresholds
feat_abund <- feature_abundances(p0)$absolute_abundance

# create a modified version of p0 with additional constraints for each
# feature to specify that the planning units in the solution must
# secure at least 8% of the total abundance for each feature
p2 <- p0
for (i in seq_len(terra::nlyr(sim_features))) {
  p2 <-
    p2 %>%
    add_linear_constraints(
      threshold = feat_abund[i] * 0.08,
      sense = ">=",
      data = sim_features[[i]]
    )
}

# overall, p2 could be described as an optimization problem
# that maximizes feature representation as much as possible
# towards securing 20% of the total amount of each feature,
# whilst ensuring that (i) the total cost of the solution does
# not exceed 1800 (per cost values in sim_pu_raster) and (ii)
# the solution secures at least 8% of the total amount of each feature
# (if 20% is not possible due to the budget)

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

```
# plot solutions s0 and s2 to compare them
plot(c(s0, s2), main = c("s1", "s2"), axes = FALSE)

# third, let's create a modified version of p0 that contains
# additional constraints to ensure that the solution equitably
# distributes conservation effort across different administrative areas
# (e.g., countries) within the study region

# to begin with, we will simulate a dataset describing the spatial extent of
# four different administrative areas across the study region
sim_admin <- sim_pu_raster
sim_admin <- terra::aggregate(sim_admin, fact = 5)
sim_admin <- terra::setValues(sim_admin, seq_len(terra::ncell(sim_admin)))
sim_admin <- terra::resample(sim_admin, sim_pu_raster, method = "near")
sim_admin <- terra::mask(sim_admin, sim_pu_raster)

# plot administrative areas layer,
# we can see that the administrative areas subdivide
# the study region into four quadrants, and the sim_admin object is a
# SpatRaster with integer values denoting ids for the administrative areas
plot(sim_admin, axes = FALSE)

# next we will convert the sim_admin SpatRaster object into a SpatRaster
# object (with a layer for each administrative area) indicating which
# planning units belong to each administrative area using binary
# (presence/absence) values
sim_admin2 <- binary_stack(sim_admin)

# plot binary stack of administrative areas
plot(sim_admin2, axes = FALSE)

# we will now calculate the total amount of planning units associated
# with each administrative area, so that we can set the constraint threshold

# since we are using raster data, we won't bother explicitly
# accounting for the total area of each planning unit (because all
# planning units have the same area in raster formats) -- but if we were
# using vector data then we would need to account for the area of each unit
admin_total <- Matrix::rowSums(rij_matrix(sim_pu_raster, sim_admin2))

# create a modified version of p0 with additional constraints for each
# administrative area to specify that the planning units in the solution must
# not encompass more than 10% of the total extent of the administrative
# area
p3 <- p0
for (i in seq_len(terra::nlyr(sim_admin2))) {
  p3 <-
    p3 %>%
    add_linear_constraints(
      threshold = admin_total[i] * 0.1,
      sense = "<=",
      data = sim_admin2[[i]]
    )
```

```
}

# solve problem
s3 <- solve(p3)

# plot solutions s0 and s3 to compare them
plot(c(s0, s3), main = c("s0", "s3"), axes = 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,SpatRaster'
add_linear_penalties(x, penalty, data)

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

Arguments

x	`problem()` object.

penalty — numeric penalty value that is used to scale the importance of 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, terra::rast(), matrix, or Matrix object containing the values 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.

Value

An updated problem() object with the penalties added to it.

Data format

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

data **as** character **vector** containing column name(s) that contain penalty values for planning units. This format is only compatible if the planning units in the argument to x are a sf::sf() or data.frame object. The column(s) must have numeric values, and must not contain any missing (NA) values. For problems that contain a single zone, the argument to data must contain a single column name. Otherwise, for problems that contain multiple zones, the argument to data must contain a column name for each zone.

data **as a** numeric **vector** containing values for planning units. These values must not contain any missing (NA) values. Note that this format is only available for planning units that contain a single zone.

data **as a** matrix/Matrix **object** containing numeric values that specify data for 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.

data **as a** terra::rast() **object** containing values for planning units. This format is only compatible if the planning units in the argument to x are sf::sf(), or terra::rast() objects. If the planning unit data are a sf::sf() object, then the values are calculated by overlaying the planning units with the argument to data and calculating the sum of the values associated with each planning unit. If the planning unit data are a terra::rast() object, then the values are calculated by extracting the cell values (note that the planning unit data and the argument to data 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.

Mathematical formulation

The linear penalties are implemented 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 indicating if each 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 $-P_z$.

See Also

See penalties for an overview of all functions for adding penalties. Also, see `calibrate_cohon_penalty()` for assistance with selecting an appropriate `penalty` value.

Other functions for adding penalties: `add_asym_connectivity_penalties()`, `add_boundary_penalties()`, `add_connectivity_penalties()`, `add_feature_weights()`, `add_neighbor_penalties()`

Examples

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

# load data
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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
plot(sim_pu_polygons[, "penalty_data"], axes = 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)

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

# create a new object with both solutions
s3 <- sf::st_sf(
  tibble::tibble(
    s1 = s1$solution_1,
    s2 = s2$solution_1
  ),
  geometry = sf::st_geometry(s1)
)


# 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
plot(s3, axes = 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

# 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_zones_features),
  nrow = number_of_features(sim_zones_features)
)

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

# plot the penalty data, each layer corresponds to a different zone
plot(penalty_raster, main = "penalty data", axes = 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
p4 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  add_min_set_objective() %>%
  add_relative_targets(targ) %>%
  add_linear_penalties(c(1, 1, 1), penalty_raster) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# print problem
print(p4)

# solve problem
s4 <- solve(p4)

# plot solution
plot(category_layer(s4), main = "multi-zone solution", axes = 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)

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

Arguments

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

Value

An updated problem() object with the constraints added to it.

Data format

The following formats can be used to lock in planning units.

locked_in **as a** numeric **vector** containing numeric values that indicate which planning units should be locked for the solution. If x has data.frame planning units, then these values must refer to values in the id column of the planning unit data. Alternatively, if x has sf::st_sf() or matrix planning units, then these values must refer to the row numbers of the planning unit data. Additionally, if x has numeric vector planning units, then these values must refer to the element indices of the planning unit data. Finally, if x has terra::rast() planning units, then these values must refer to cell indices. Note that this format is available for problems that contain a single zone.

locked_in **as a** logical **vector** containing TRUE and/or FALSE values that indicate each if planning units should be locked in the solution. Note that the vector should have a TRUE or FALSE value for each and every planning unit in the argument to x. This argument is only compatible with problems that contain a single zone.

locked_in **as a** matrix **object** containing logical (i.e., TRUE or FALSE) values that indicate if certain planning units 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.

locked_in **as a** character **vector** containing column name(s) for the planning unit data in x that indicate if planning units should be locked for the solution. This format is only compatible if

the argument to x has sf::st_sf() or data.frame planning units. The columns must have logical (i.e., TRUE or FALSE) values indicating if planning units should be locked for the solution. For problems that contain a single zone, the argument to data must contain a single column name. Otherwise, for problems that contain multiple zones, the argument to data must contain a column name for each zone.

locked_in **as a** sf::sf() **object** containing geometries that will be used to lock planning units for the solution. Specifically, planning units in x that spatially intersect with y will be locked (per intersecting_units()). Note that this option is only available for problems that contain a single management zone.

locked_in **as a** terra::rast() **object** containing cells used to lock planning units for the solution. Specifically, planning units in x that intersect with cells that have non-zero and non-NA values are locked. For problems that contain multiple zones, the data object must contain a layer for each zone. Note that for multi-band arguments, each cell must only contain a non-zero value in a single band. Additionally, if the cost data in x is a terra::rast() object, we recommend standardizing NA values in this dataset with the cost data. In other words, the cells in x that have NA values should also have NA values in the locked data.

See Also

See constraints for an overview of all functions for adding constraints.

Other functions for adding constraints: add_contiguity_constraints(), add_feature_contiguity_constraints(), add_linear_constraints(), add_locked_out_constraints(), add_mandatory_allocation_constraints(), add_manual_bounded_constraints(), add_manual_locked_constraints(), add_neighbor_constraints()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()
sim_locked_in_raster <- get_sim_locked_in_raster()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_pu_polygons <- get_sim_zones_pu_polygons()
sim_zones_features <- get_sim_zones_features()

# 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 column 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)

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

# create single object with all solutions
s6 <- sf::st_sf(
  tibble::tibble(
    s1 = s1$solution_1,
    s2 = s2$solution_1,
    s3 = s3$solution_1,
    s4 = s4$solution_1,
    s5 = s5$solution_1
  ),
  geometry = sf::st_geometry(s1)
)

# plot solutions
plot(
  s6,
  main = c(
    "none locked in", "locked in (integer input)",
    "locked in (character input)", "locked in (raster input)",
    "locked in (polygon input)"
  )
)

# create minimal multi-zone problem with spatial data
p7 <-
  problem(
    sim_zones_pu_polygons, sim_zones_features,
    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 <- as.matrix(sf::st_drop_geometry(
  sim_zones_pu_polygons[, c("locked_1", "locked_2", "locked_3")]
))
```

```
p8 <- p7 %>% add_locked_in_constraints(locked_matrix)

# solve problem
s8 <- solve(p8)

# create new column representing the zone id that each planning unit
# was allocated to in the solution
s8$solution <- category_vector(sf::st_drop_geometry(
  s8[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")]
))
s8$solution <- factor(s8$solution)

# plot solution
plot(s8[ "solution"], axes = FALSE)

# create multi-zone problem with locked in constraints using column names
p9 <- p7 %>% add_locked_in_constraints(c("locked_1", "locked_2", "locked_3"))

# solve problem
s9 <- solve(p9)

# create new column representing the zone id that each planning unit
# was allocated to in the solution
s9$solution <- category_vector(sf::st_drop_geometry(
  s9[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")]
))
s9$solution[s9$solution == 1 & s9$solution_1_zone_1 == 0] <- 0
s9$solution <- factor(s9$solution)

# plot solution
plot(s9[, "solution"], axes = FALSE)

# create multi-zone problem with raster planning units
p10 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  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-layer raster with locked in units
locked_in_raster <- sim_zones_pu_raster[[1]]
locked_in_raster[!is.na(locked_in_raster)] <- 0
locked_in_raster <- locked_in_raster[[c(1, 1, 1)]]
names(locked_in_raster) <- c("zone_1", "zone_2", "zone_3")
locked_in_raster[[1]][1] <- 1
locked_in_raster[[2]][2] <- 1
locked_in_raster[[3]][3] <- 1

# plot locked in raster
plot(locked_in_raster)
```

```
# add locked in raster units to problem
p10 <- p10 %>% add_locked_in_constraints(locked_in_raster)

# solve problem
s10 <- solve(p10)

# plot solution
plot(category_layer(s10), main = "solution", axes = 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_in_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)

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

Arguments

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

Value

An updated problem() object with the constraints added to it.

Data format

The following formats can be used to lock in planning units.

locked_out **as a** numeric **vector** containing numeric values that indicate which planning units should be locked for the solution. If x has data.frame planning units, then these values must refer to values in the id column of the planning unit data. Alternatively, if x has sf::st_sf() or matrix planning units, then these values must refer to the row numbers of the planning unit data. Additionally, if x has numeric vector planning units, then these values must refer to the element indices of the planning unit data. Finally, if x has terra::rast() planning units, then these values must refer to cell indices. Note that this format is available for problems that contain a single zone.

locked_out **as a** logical **vector** containing TRUE and/or FALSE values that indicate each if planning units should be locked in the solution. Note that the vector should have a TRUE or FALSE value for each and every planning unit in the argument to x. This argument is only compatible with problems that contain a single zone.

locked_out **as a** matrix **object** containing logical (i.e., TRUE or FALSE) values that indicate if certain planning units 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.

locked_out **as a** character **vector** containing column name(s) for the planning unit data in x that indicate if planning units should be locked for the solution. This format is only compatible if the argument to x has sf::st_sf() or data.frame planning units. The columns must have logical (i.e., TRUE or FALSE) values indicating if planning units should be locked for the solution. For problems that contain a single zone, the argument to data must contain a single column name. Otherwise, for problems that contain multiple zones, the argument to data must contain a column name for each zone.

locked_out **as a** sf::sf() **object** containing geometries that will be used to lock planning units for the solution. Specifically, planning units in x that spatially intersect with y will be locked (per intersecting_units()). Note that this option is only available for problems that contain a single management zone.

locked_out **as a** terra::rast() **object** containing cells used to lock planning units for the solution. Specifically, planning units in x that intersect with cells that have non-zero and non-NA values are locked. For problems that contain multiple zones, the data object must contain a layer for each zone. Note that for multi-band arguments, each cell must only contain a non-zero value in a single band. Additionally, if the cost data in x is a terra::rast() object, we recommend standardizing NA values in this dataset with the cost data. In other words, the cells in x that have NA values should also have NA values in the locked data.

See Also

See constraints for an overview of all functions for adding constraints.

Other functions for adding constraints: add_contiguity_constraints(), add_feature_contiguity_constraints(), add_linear_constraints(), add_locked_in_constraints(), add_mandatory_allocation_constraints(), add_manual_bounded_constraints(), add_manual_locked_constraints(), add_neighbor_constraints()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()
sim_locked_out_raster <- get_sim_locked_out_raster()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_pu_polygons <- get_sim_zones_pu_polygons()
sim_zones_features <- get_sim_zones_features()

# 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 column 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)

# solve problems
```

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

# create single object with all solutions
s6 <- sf::st_sf(
  tibble::tibble(
    s1 = s1$solution_1,
    s2 = s2$solution_1,
    s3 = s3$solution_1,
    s4 = s4$solution_1,
    s5 = s5$solution_1
  ),
  geometry = sf::st_geometry(s1)
)

# plot solutions
plot(
  s6,
  main = c(
    "none locked out", "locked out (integer input)",
    "locked out (character input)", "locked out (raster input)",
    "locked out (polygon input)"
  )
)

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

# create minimal multi-zone problem with spatial data
p7 <-
  problem(
    sim_zones_pu_polygons, sim_zones_features,
    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 <- as.matrix(sf::st_drop_geometry(
  sim_zones_pu_polygons[, c("locked_1", "locked_2", "locked_3")]
))

p8 <- p7 %>% add_locked_out_constraints(locked_matrix)

# solve problem
s8 <- solve(p8)

# create new column representing the zone id that each planning unit
```

```
# was allocated to in the solution
s8$solution <- category_vector(sf::st_drop_geometry(
  s8[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")]
))
s8$solution <- factor(s8$solution)

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

# create multi-zone problem with locked out constraints using column names
p9 <-
  p7 %>%
  add_locked_out_constraints(c("locked_1", "locked_2", "locked_3"))

# solve problem
s9 <- solve(p9)

# create new column in s8 representing the zone id that each planning unit
# was allocated to in the solution
s9$solution <- category_vector(sf::st_drop_geometry(
  s9[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")]
))
s9$solution[s9$solution == 1 & s9$solution_1_zone_1 == 0] <- 0
s9$solution <- factor(s9$solution)

# plot solution
plot(s9[, "solution"], main = "solution", axes = FALSE)

# create multi-zone problem with raster planning units
p10 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  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-layer raster with locked out units
locked_out_raster <- sim_zones_pu_raster[[1]]
locked_out_raster[!is.na(locked_out_raster)] <- 0
locked_out_raster <- locked_out_raster[[c(1, 1, 1)]]
names(locked_out_raster) <- c("zones_1", "zones_2", "zones_3")
locked_out_raster[[1]][1] <- 1
locked_out_raster[[2]][2] <- 1
locked_out_raster[[3]][3] <- 1

# plot locked out raster
plot(locked_out_raster)

# add locked out raster units to problem
p10 <- p10 %>% add_locked_out_constraints(locked_out_raster)

# solve problem
s10 <- solve(p10)
```

```
# plot solution
plot(category_layer(s10), main = "solution", axes = FALSE)

## End(Not run)
```

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

Description

Specify that the *SYMPHONY* software – using the **lpsymphony** package – should be used to solve a conservation planning problem (Ralphs & Güzelsoy 2005). 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()` 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

An updated `problem()` object 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

Other functions for adding solvers: `add_cbc_solver()`, `add_cplex_solver()`, `add_default_solver()`, `add_gurobi_solver()`, `add_highs_solver()`, `add_rsymphony_solver()`

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_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)

## End(Not run)
```

`add_mandatory_allocation_constraints` *Add mandatory allocation constraints*

Description

Add constraints to a conservation planning problem 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

```
add_mandatory_allocation_constraints(x)
```

Arguments

x	`problem()` 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 planning unit to a zone. For example, when developing land-use plans, some decision makers may require that every parcel of land is 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 are 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

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

See Also

See constraints for an overview of all functions for adding constraints.

Other functions for adding constraints: `add_contiguity_constraints()`, `add_feature_contiguity_constraints()`, `add_linear_constraints()`, `add_locked_in_constraints()`, `add_locked_out_constraints()`, `add_manual_bounded_constraints()`, `add_manual_locked_constraints()`, `add_neighbor_constraints()`

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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_zones_pu_raster, sim_zones_features) %>%
  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()

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

# convert solutions into category layers, where each cell 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(c(c1, c2), main = c("default", "mandatory allocation"), axes = FALSE)

## End(Not run)
```

---

add_manual_bounded_constraints

Add manually specified bound 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() object.
data	data.frame or tibble::tibble() object. See the Data format section for more information.

Value

An updated problem() object with the constraints added to it.

Data format

The argument to data should be a data.frame with the following columns:

pu integer planning unit identifiers. If x has data.frame planning units, then these values must refer to values in the id column of the planning unit data. Alternatively, if x has sf::st_sf() or matrix planning units, then these values must refer to the row numbers of the planning unit data. Additionally, if x has numeric vector planning units, then these values must refer to the element indices of the planning unit data. Finally, if x has terra::rast() planning units, then these values must refer to cell indices.

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

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

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

See Also

See constraints for an overview of all functions for adding constraints.

Other functions for adding constraints: add_contiguity_constraints(), add_feature_contiguity_constraints(), add_linear_constraints(), add_locked_in_constraints(), add_locked_out_constraints(), add_mandatory_allocation_constraints(), add_manual_locked_constraints(), add_neighbor_constraints()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)
```

```
# load data
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()
sim_zones_pu_polygons <- get_sim_zones_pu_polygons()
sim_zones_features <- get_sim_zones_features()

# 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)

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

# create object with all solutions
s4 <- sf::st_sf(
  tibble::tibble(
    s1 = s1$solution_1,
    s2 = s2$solution_1,
    s3 = s3$solution_1
  ),
  geometry = sf::st_geometry(s1)
)

# plot solutions
## s1 = none locked in
## s2 = locked in constraints
## s3 = manual bounds constraints
plot(s4)

# create minimal problem with multiple zones
p5 <-
  problem(
    sim_zones_pu_polygons, sim_zones_features,
    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
p6 <- p5 %>% add_manual_bounded_constraints(bounds_data2)

# solve problem
s5 <- solve(p5)
s6 <- solve(p6)

# create two new columns representing the zone id that each planning unit
# was allocated to in the two solutions
s5$solution <- category_vector(sf::st_drop_geometry(
  s5[, c("solution_1_zone_1","solution_1_zone_2", "solution_1_zone_3")]
))
s5$solution <- factor(s5$solution)

s5$solution_bounded <- category_vector(sf::st_drop_geometry(
  s6[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")]
))
s5$solution_bounded <- factor(s5$solution_bounded)

# plot solutions
plot(s5[, c("solution", "solution_bounded")], axes = 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()` object.
`data`	`data.frame` or `tibble::tibble()` object. See the Data format section for more information.

Value

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

Data format

The argument to `data` should be a `data.frame` with the following columns:

pu `integer` planning unit identifiers. If `x` has `data.frame` planning units, then these values must refer to values in the `id` column of the planning unit data. Alternatively, if `x` has `sf::st_sf()` or `matrix` planning units, then these values must refer to the row numbers of the planning unit data. Additionally, if `x` has `numeric` vector planning units, then these values must refer to the element indices of the planning unit data. Finally, if `x` has `terra::rast()` planning units, then these values must refer to cell indices.

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

status `numeric` status values. These values indicate 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

See constraints for an overview of all functions for adding constraints.

Other functions for adding constraints: add_contiguity_constraints(), add_feature_contiguity_constraints(), add_linear_constraints(), add_locked_in_constraints(), add_locked_out_constraints(), add_mandatory_allocation_constraints(), add_manual_bounded_constraints(), add_neighbor_constraints()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()
sim_zones_pu_polygons <- get_sim_zones_pu_polygons()
sim_zones_features <- get_sim_zones_features()

# 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)

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

# create object with all solutions
s4 <- sf::st_sf(
  tibble::tibble(
    s1 = s1$solution_1,
    s2 = s2$solution_1,
    s3 = s3$solution_1
  ),
  geometry = sf::st_geometry(s1)
)
```

```
# plot solutions
## s1 = none locked in
## s2 = locked in constraints
## s3 = manual locked constraints
plot(s4)

# create minimal problem with multiple zones
p5 <-
  problem(
    sim_zones_pu_polygons, sim_zones_features,
    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
p6 <- p5 %>% add_manual_locked_constraints(locked_data2)

# solve problem
s5 <- solve(p5)
s6 <- solve(p6)

# create two new columns representing the zone id that each planning unit
# was allocated to in the two solutions
s5$solution <- category_vector(sf::st_drop_geometry(
  s5[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")]
))
s5$solution <- factor(s5$solution)

s5$solution_locked <- category_vector(sf::st_drop_geometry(
  s6[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")]
))
s5$solution_locked <- factor(s5$solution_locked)

# plot solutions
plot(s5[, c("solution", "solution_locked")], axes = FALSE)
```

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

`add_manual_targets` *Add manual targets*

Description

Add targets to 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()` object.
`targets`	`data.frame` or `tibble::tibble()` object. See the Targets format section for more information.

Details

This function is used to set targets for each feature (separately). For problems associated with a single management zone, this function may be useful to specify individual targets for each feature. For problems associated with multiple management zones, this function can also be used to specify a target for each feature within each zone (separately). For example, this may be useful in planning exercises where it is important to ensure that some of the features are adequately represented by multiple zones. For example, in a marine spatial planning exercise, it may be important for some features (e.g., commercial important fish species) to be adequately represented by a conservation zone for ensuring their long-term persistence, and also by a fishing zone to for ensure food security. For greater flexibility in target setting (such as setting targets that can be met through the allocation of multiple zones), see the `add_manual_targets()` function.

Value

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

Targets format

The `targets` argument should be a `data.frame` with the following columns:

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

zone `character` name of zones in the argument `x`. It can also be a `list` of `character` vectors if targets should correspond to multiple zones (see Examples section below). This 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 column is optional and if it is missing then target senses will default to `">="` values.

target `numeric` target threshold.

Target setting

Many conservation planning problems require targets. Targets are used to specify the minimum amount, or proportion, of a feature's spatial distribution that should ideally be protected. This is important so that the optimization process can weigh the merits and trade-offs between improving the representation of one feature over another feature. Although it can be challenging to set meaningful targets, this is a critical step for ensuring that prioritizations meet the stakeholder objectives that underpin a prioritization exercise (Carwardine *et al.* 2009). In other words, targets play an important role in ensuring that a priority setting process is properly tuned according to stakeholder requirements. For example, targets provide a mechanism for ensuring that a prioritization secures enough habitat to promote the long-term persistence of each threatened species, culturally important species, or economically important ecosystem services under consideration. Since there is often uncertainty regarding stakeholder objectives (e.g., how much habitat should be protected for a given species) or the influence of particular target on a prioritization (e.g., how would setting a 90% or 100% for a threatened species alter priorities), it is often useful to generate and compare a suite of prioritizations based on different target scenarios.

References

Carwardine J, Klein CJ, Wilson KA, Pressey RL, Possingham HP (2009) Hitting the target and missing the point: target-based conservation planning in context. *Conservation Letters*, 2: 4–11.

See Also

See targets for an overview of all functions for adding targets.

Other functions for adding targets: `add_absolute_targets()`, `add_auto_targets()`, `add_group_targets()`, `add_relative_targets()`

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)
```

```
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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)

# solve problem
s1 <- solve(p1)

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

# 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)

# solve problem
s2 <- solve(p2)

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

# 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)

# solve problem
s3 <- solve(p3)

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

# 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)

# solve problem
s4 <- solve(p4)

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

# 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_zones_pu_raster, sim_zones_features) %>%
  add_min_set_objective() %>%
  add_absolute_targets(targets_matrix) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s5 <- solve(p5)

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

# create equivalent problem using add_manual_targets
targets_dataframe <- expand.grid(
  feature = feature_names(sim_zones_features),
  zone = zone_names(sim_zones_features),
```

```
  sense = ">=",
  type = "absolute"
)
targets_dataframe$target <- c(targets_matrix)

p6 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  add_min_set_objective() %>%
  add_manual_targets(targets_dataframe) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s6 <- solve(p6)

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

# 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_zones_features)[[1]][[1:2]]
spp_zone2 <- spp_zone1 * 0.5
costs <- sim_zones_pu_raster[[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)
```

```
# solve problem
s7 <- solve(p7)

# plot solution
plot(category_layer(s7), main = "solution", axes = 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 objective does not use targets, and feature weights should be used instead to increase the representation of certain features by a solution.

Usage

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

Arguments

`x`	`problem()` 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

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.

Value

An updated `problem()` object with the objective added to it.

Mathematical formulation

This objective is based on the maximum coverage reserve selection problem (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:

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

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.

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

See objectives for an overview of all functions for adding objectives. Also, see `add_feature_weights()` to specify weights for different features.

Other functions for adding objectives: `add_max_features_objective()`, `add_max_phylo_div_objective()`, `add_max_phylo_end_objective()`, `add_max_utility_objective()`, `add_min_largest_shortfall_objective()`, `add_min_penalties_objective()`, `add_min_set_objective()`, `add_min_shortfall_objective()`

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_features <- get_sim_features()
sim_zones_features <- get_sim_zones_features()
```

```
# threshold the feature data to generate binary biodiversity data
sim_binary_features <- sim_features
thresholds <- terra::global(
  sim_features, fun = quantile, probs = 0.5, na.rm = TRUE
)
for (i in seq_len(terra::nlyr(sim_features))) {
  sim_binary_features[[i]] <- terra::as.int(
    sim_features[[i]] > thresholds[[1]][[i]]
  )
}

# create problem with maximum cover objective
p1 <-
  problem(sim_pu_raster, sim_binary_features) %>%
  add_max_cover_objective(500) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s1 <- solve(p1)

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

# threshold the multi-zone feature data to generate binary biodiversity data
sim_binary_features_zones <- sim_zones_features
for (z in seq_len(number_of_zones(sim_zones_features))) {
  thresholds <- terra::global(
    sim_zones_features[[z]], fun = quantile, probs = 0.5, na.rm = TRUE
  )
  for (i in seq_len(number_of_features(sim_zones_features))) {
    sim_binary_features_zones[[z]][[i]] <- terra::as.int(
      sim_zones_features[[z]][[i]] > thresholds[[1]][[i]]
    )
  }
}

# create multi-zone problem with maximum cover objective that
# has a single budget for all zones
p2 <-
  problem(sim_zones_pu_raster, sim_binary_features_zones) %>%
  add_max_cover_objective(800) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s2 <- solve(p2)

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

# create multi-zone problem with maximum cover objective that
# has separate budgets for each zone
```

```
p3 <-
  problem(sim_zones_pu_raster, sim_binary_features_zones) %>%
  add_max_cover_objective(c(400, 400, 400)) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s3 <- solve(p3)

# plot solution
plot(category_layer(s3), main = "solution", axes = 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, whilst ensuring that the cost of the solution does not exceed a budget.

Usage

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

Arguments

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

Details

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.

Value

An updated `problem()` object with the objective added to it.

Mathematical formulation

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:

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

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.

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

See objectives for an overview of all functions for adding objectives. Also, see targets for an overview of all functions for adding targets, and `add_feature_weights()` to specify weights for different features.

Other functions for adding objectives: `add_max_cover_objective()`, `add_max_phylo_div_objective()`, `add_max_phylo_end_objective()`, `add_max_utility_objective()`, `add_min_largest_shortfall_objective()`, `add_min_penalties_objective()`, `add_min_set_objective()`, `add_min_shortfall_objective()`

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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)

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

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

# 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_zones_pu_raster, sim_zones_features) %>%
  add_max_features_objective(3000) %>%
  add_relative_targets(matrix(0.1, ncol = 3, nrow = 5)) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s2 <- solve(p2)

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

# 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_zones_pu_raster, sim_zones_features) %>%
  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)

# solve problem
s3 <- solve(p3)

# plot solution
plot(category_layer(s3), main = "solution", axes = 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() object.
budget	numeric value specifying the maximum expenditure of the prioritization. For problems with multiple zones, the argument to budget can be (i) a single numeric value to specify a single budget for the entire solution or (ii) a numeric vector to specify a separate budget for each management zone.
tree	ape::phylo() object specifying a phylogenetic tree for the conservation features.

Details

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.

Value

An updated problem() object with the objective added to it.

Mathematical formulation

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:

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

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. 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.

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

See objectives for an overview of all functions for adding objectives. Also, see targets for an overview of all functions for adding targets, and `add_feature_weights()` to specify weights for different features.

Other functions for adding objectives: `add_max_cover_objective()`, `add_max_features_objective()`, `add_max_phylo_end_objective()`, `add_max_utility_objective()`, `add_min_largest_shortfall_objective()`, `add_min_penalties_objective()`, `add_min_set_objective()`, `add_min_shortfall_objective()`

Examples

```
## Not run:
# load ape package
require(ape)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_phylogeny <- get_sim_phylogeny()
sim_zones_pu_raster  <- get_sim_zones_pu_raster()
sim_zones_features  <- get_sim_zones_features()

# plot the simulated phylogeny
par(mfrow = c(1, 1))
plot(sim_phylogeny, main = "phylogeny")

# 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)

# solve problem
s1 <- solve(p1)

# plot solution
plot(s1, main = "solution", axes = 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", terra::nlyr(sim_features)),
    sim_phylogeny$tip.label %in% r1$feature[r1$met], "red"
  )
)

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

# 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_zones_features),
  zone = list(zone_names(sim_zones_features))[
    rep(1, number_of_features(sim_zones_features))],
  type = rep("absolute", number_of_features(sim_zones_features)),
  target = rep(10, number_of_features(sim_zones_features))
)

# create a multi-zone problem with a maximum phylogenetic diversity
# objective, where the total expenditure in all zones is 5000.
p2 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  add_max_phylo_div_objective(5000, sim_phylogeny) %>%
  add_manual_targets(targets) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s2 <- solve(p2)

# plot solution
plot(category_layer(s2), main = "solution", axes = 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", terra::nlyr(sim_features)), which(r2$met), "red"
  )
)

# create a multi-zone problem with a maximum phylogenetic diversity
# objective, where each zone has a separate budget.
p3 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  add_max_phylo_div_objective(c(2500, 500, 2000), sim_phylogeny) %>%
  add_manual_targets(targets) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s3 <- solve(p3)

# plot solution
plot(category_layer(s3), main = "solution", axes = 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", terra::nlyr(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_div_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() object.
budget	numeric value specifying the maximum expenditure of the prioritization. For problems with multiple zones, the argument to budget can be (i) a single numeric value to specify a single budget for the entire solution or (ii) a numeric vector to specify a separate budget for each management zone.
tree	ape::phylo() object specifying a phylogenetic tree for the conservation features.

Details

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.

Value

An updated problem() object with the objective added to it.

Mathematical formulation

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:

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

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.

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

See objectives for an overview of all functions for adding objectives. Also, see targets for an overview of all functions for adding targets, and `add_feature_weights()` to specify weights for different features.

Other functions for adding objectives: `add_max_cover_objective()`, `add_max_features_objective()`, `add_max_phylo_div_objective()`, `add_max_utility_objective()`, `add_min_largest_shortfall_objective()`, `add_min_penalties_objective()`, `add_min_set_objective()`, `add_min_shortfall_objective()`

Examples

```
## Not run:
# load ape package
require(ape)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_phylogeny <- get_sim_phylogeny()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()
```

```
# plot the simulated phylogeny
par(mfrow = c(1, 1))
plot(sim_phylogeny, main = "phylogeny")

# 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)

# solve problem
s1 <- solve(p1)

# plot solution
plot(s1, main = "solution", axes = 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", terra::nlyr(sim_features)),
    sim_phylogeny$tip.label %in% r1$feature[r1$met],
    "red"
  )
)

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

# 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_zones_features),
  zone = list(zone_names(sim_zones_features))[
    rep(1, number_of_features(sim_zones_features))],
  type = rep("absolute", number_of_features(sim_zones_features)),
  target = rep(10, number_of_features(sim_zones_features))
)

# create a multi-zone problem with a maximum phylogenetic endemism
# objective, where the total expenditure in all zones is 5000.
p2 <-
```

```
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  add_max_phylo_end_objective(5000, sim_phylogeny) %>%
  add_manual_targets(targets) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s2 <- solve(p2)

# plot solution
plot(category_layer(s2), main = "solution", axes = 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", terra::nlyr(sim_features)), which(r2$met), "red"
  )
)

# create a multi-zone problem with a maximum phylogenetic endemism
# objective, where each zone has a separate budget.
p3 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  add_max_phylo_end_objective(c(2500, 500, 2000), sim_phylogeny) %>%
  add_manual_targets(targets) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s3 <- solve(p3)

# plot solution
plot(category_layer(s3), main = "solution", axes = 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", terra::nlyr(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 maximize the weighted sum of the features represented by the solution as much as possible without exceeding a budget. This objective does not use targets, and feature weights should be used instead to increase the representation of particular features by a solution. Note that this objective does not aim to maximize as much of each feature individually, and so often results in solutions that are heavily biased towards just a few features.

Usage

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

Arguments

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

Details

The maximum utility objective seeks to maximize 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 particular features over other features (see `add_feature_weights()`). It is essentially calculated as a weighted sum of the feature data inside the selected planning units.

Value

An updated `problem()` object with the objective added to it.

Mathematical formulation

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.

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

See objectives for an overview of all functions for adding objectives. Also, see add_feature_weights() to specify weights for different features.

Other functions for adding objectives: add_max_cover_objective(), add_max_features_objective(), add_max_phylo_div_objective(), add_max_phylo_end_objective(), add_min_largest_shortfall_objective(), add_min_penalties_objective(), add_min_set_objective(), add_min_shortfall_objective()

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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)

# solve problem
s1 <- solve(p1)

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

# create multi-zone problem with maximum utility objective that
# has a single budget for all zones
p2 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  add_max_utility_objective(5000) %>%
  add_binary_decisions() %>%
  add_default_solver(gap = 0, verbose = FALSE)

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

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

# create multi-zone problem with maximum utility objective that
# has separate budgets for each zone
p3 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  add_max_utility_objective(c(1000, 2000, 3000)) %>%
  add_binary_decisions() %>%
  add_default_solver(gap = 0, verbose = FALSE)

# solve problem
s3 <- solve(p3)

# plot solution
plot(category_layer(s3), main = "solution", axes = 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()` object.
budget	`numeric` value specifying the maximum expenditure of the prioritization. For problems with multiple zones, the argument to `budget` can be (i) a single `numeric` value to specify a single budget for the entire solution or (ii) a `numeric` vector to specify a separate budget for each management zone.

Details

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()).

Value

An updated problem() object with the objective added to it.

Mathematical formulation

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:

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

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, and t_j is the representation target for feature j. Additionally, l denotes the largest relative target shortfall among all the species. Furthermore, B is the budget allocated for the solution, c_i is the cost of planning unit i. Note that continuous variable l is bounded between zero and one.

See Also

See objectives for an overview of all functions for adding objectives. Also, see targets for an overview of all functions for adding targets, and add_feature_weights() to specify weights for different features.

Other functions for adding objectives: add_max_cover_objective(), add_max_features_objective(), add_max_phylo_div_objective(), add_max_phylo_end_objective(), add_max_utility_objective(), add_min_penalties_objective(), add_min_set_objective(), add_min_shortfall_objective()

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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)

# solve problem
```

```
s1 <- solve(p1)

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

# 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 1800
p2 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  add_min_largest_shortfall_objective(1800) %>%
  add_relative_targets(matrix(0.1, ncol = 3, nrow = 5)) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s2 <- solve(p2)

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

# create multi-zone problem with minimum largest shortfall objective,
# with 10% representation targets for each feature, and set
# separate budgets of 1800 for each management zone
p3 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  add_min_largest_shortfall_objective(c(1800, 1800, 1800)) %>%
  add_relative_targets(matrix(0.1, ncol = 3, nrow = 5)) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s3 <- solve(p3)

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

## End(Not run)
```

`add_min_penalties_objective` *Add minimum penalties objective*

Description

Set the objective of a conservation planning problem to only minimize the penalties added to the problem, whilst ensuring that all targets are met and the cost of the solution does not exceed a budget. This objective is useful when using a hierarchical approach for multi-objective optimization.

Usage

```
add_min_penalties_objective(x, budget)
```

Arguments

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

Details

The minimum penalty objective is designed to be used with problems that have penalties (see penalties for details). It can be used to generate solutions that focus entirely on minimizing the penalties, whilst ensuring that certain constraints are met. This is is useful when performing multi-objective optimization using a hierarchical approach (see examples below). Although previous versions of the package recommended using the minimum set objective (i.e., add_min_set_objective()) with zero costs and linear constraints for this purpose, the minimum penalty objective provides a dedicated objective for performing hierarchical multi-objective optimization.

Value

An updated problem() object with the objective added to it.

Mathematical formulation

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:

$$Minimize 0 subject to \sum_{i=1}^{I} x_i r_{ij} \geq T_j \forall j \in J \sum_{i=1}^{I} x_i c_i \leq B$$

Here, x_i is the decisions variable (e.g., specifying whether planning unit i has been selected (1) or not (0)), 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. Since the objective is to minimize zero, this function does not actually provide any criteria to compare competing solutions. As such, when used in conjunction with a penalty function (see penalties), only the penalty (e.g., add_boundary_penalties()) will be used to compare competing solutions during optimization.

See Also

See objectives for an overview of all functions for adding objectives. Also see targets for an overview of all functions for adding targets. Additionally, see penalties for an overview of all functions for adding penalties.

Other functions for adding objectives: add_max_cover_objective(), add_max_features_objective(), add_max_phylo_div_objective(), add_max_phylo_end_objective(), add_max_utility_objective(), add_min_largest_shortfall_objective(), add_min_set_objective(), add_min_shortfall_objective()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# create initial 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)

# solve initial problem
s1 <- solve(p1)

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

# calculate total cost of initial solution
c1 <- eval_cost_summary(p1, s1)

# since the solution is spatially fragmented, we will now use
# a hierarchical multi-objective optimization approach to reduce
# spatial fragmentation

# calculate budget for new prioritization based on cost of initial solution,
# this budget will specify that we are willing to accept a 10%
# increase in the total cost of the solution to minimize fragmentation
b <- c1$cost[[1]] * 1.1

# create problem with minimum penalty objective using the budget and
# boundary penalties to reduce spatial fragmentation
#
# note that although we use a penalty value of 0.01 as a placeholder,
# any penalty value would give the same result since the optimization
# process is focused entirely on the boundary penalties
p2 <-
  problem(sim_pu_raster, sim_features) %>%
  add_min_penalties_objective(budget = b) %>%
  add_boundary_penalties(penalty = 0.001) %>%
  add_relative_targets(0.1) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem with minimum penalty objective
```

```
s2 <- solve(p2)

# plot solution with minimum penalty objective
plot(s2, main = "solution", axes = 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()` object.

Details

The minimum set objective – in the the context of systematic reserve design – 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 difference between this objective and the *Marxan* software is that the targets for the features will always be met (and as such it does not use Species Penalty Factors).

Value

An updated `problem()` object with the objective added to it.

Mathematical formulation

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:

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

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.

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

See objectives for an overview of all functions for adding objectives. Also see targets for an overview of all functions for adding targets.

Other functions for adding objectives: `add_max_cover_objective()`, `add_max_features_objective()`, `add_max_phylo_div_objective()`, `add_max_phylo_end_objective()`, `add_max_utility_objective()`, `add_min_largest_shortfall_objective()`, `add_min_penalties_objective()`, `add_min_shortfall_objective()`

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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)

# solve problem
s1 <- solve(p1)

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

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

p2 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  add_min_set_objective() %>%
  add_absolute_targets(targets_matrix) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

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

```
# plot solution
plot(category_layer(s2), main = "solution", axes = 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()` object.
`budget`	`numeric` value specifying the maximum expenditure of the prioritization. For problems with multiple zones, the argument to `budget` can be (i) a single `numeric` value to specify a single budget for the entire solution or (ii) a `numeric` vector to specify a separate budget for each management zone.

Details

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()`.

Value

An updated `problem()` object with the objective added to it.

Mathematical formulation

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:

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

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 relative 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 one, and denotes the relative shortfall for target j.

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

See objectives for an overview of all functions for adding objectives. Also, see targets for an overview of all functions for adding targets, and add_feature_weights() to specify weights for different features.

Other functions for adding objectives: add_max_cover_objective(), add_max_features_objective(), add_max_phylo_div_objective(), add_max_phylo_end_objective(), add_max_utility_objective(), add_min_largest_shortfall_objective(), add_min_penalties_objective(), add_min_set_objective()

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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)

# solve problem
s1 <- solve(p1)

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

# 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_zones_pu_raster, sim_zones_features) %>%
  add_min_shortfall_objective(3000) %>%
```

```
  add_relative_targets(matrix(0.1, ncol = 3, nrow = 5)) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s2 <- solve(p2)

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

# 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_zones_pu_raster, sim_zones_features) %>%
  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)

# solve problem
s3 <- solve(p3)

# plot solution
plot(category_layer(s3), main = "solution", axes = 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,ANY'
add_neighbor_constraints(x, k, clamp, zones, data)

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

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

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

Arguments

x	problem() 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.
clamp	logical should the minimum number of neighbors for selected planning units in the solution be clamped to feasibility? For example, if a planning unit has two neighbors, k = 3, and clamp = FALSE, then the planning unit could not ever be selected in the solution. However, if clamp = TRUE, then the planning unit could potentially be selected in the solution if both of its two neighbors were also selected. Defaults to TRUE.
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 to 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

An updated problem() object with the constraints added to it.

Data format

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

data **as a** NULL **value** 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).

data **as a** matrix**/**Matrix **object** 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` **as a** `data.frame` **object** containing columns that are named `"id1"`, `"id2"`, and `"boundary"`. Here, each row denotes the connectivity between two planning units following the *Marxan* format. The `"boundary"` column should contain binary `numeric` values that indicate if the two planning units specified in the `"id1"` and `"id2"` columns 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 `"zone1"` and `"zone2"` columns can optionally be provided to manually specify if the neighborhood data pertain to specific zones. The `"zone1"` and `"zone2"` columns should contain the `character` names of the zones. If the columns `"zone1"` and `"zone2"` are present, then the argument to `zones` must be `NULL`.

`data` **as an** `array` **object** 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

Other functions for adding constraints: `add_contiguity_constraints()`, `add_feature_contiguity_constraints()`, `add_linear_constraints()`, `add_locked_in_constraints()`, `add_locked_out_constraints()`, `add_mandatory_allocation_constraints()`, `add_manual_bounded_constraints()`, `add_manual_locked_constrain`

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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)
  )

# solve problems
s1 <- c(solve(p1), solve(p2), solve(p3), solve(p4))
names(s1) <- c("basic solution", "1 neighbor", "2 neighbors", "3 neighbors")

# plot solutions
plot(s1, axes = FALSE)

# create minimal problem with multiple zones
p5 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  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), zones = 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), zones = 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), zones = z8)

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

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

## End(Not run)
```

`add_neighbor_penalties` *Add neighbor penalties*

Description

Add penalties to a conservation planning problem to penalize solutions that have few neighboring planning units. These penalties can be used to promote spatial clustering in solutions. In particular, they are recommended for reducing spatial fragmentation in large-scale problems or when using open source solvers.

Usage

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

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

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

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

Arguments

x	`problem()` 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 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 adds penalties to conservation planning problem to penalize solutions that have low spatial clustering. Specifically, it favors pair-wise connections between planning units that have high connectivity values (based on Önal and Briers 2002).

Value

An updated `problem()` object with the penalties added to it.

Mathematical formulation

The neighbor penalties are implemented 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.

Data format

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

`data` **as a** `NULL` **value** 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).

`data` **as a** `matrix`/`Matrix` **object** 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` **as a** `data.frame` **object** containing columns that are named `"id1"`, `"id2"`, and `"boundary"`. Here, each row denotes the connectivity between two planning units following the *Marxan* format. The `"boundary"` column should contain binary `numeric` values that indicate if the two planning units specified in the `"id1"` and `"id2"` columns 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 `"zone1"` and `"zone2"` columns can optionally be provided to manually specify if the neighborhood data pertain to specific zones. The `"zone1"` and `"zone2"` columns should contain the `character` names of the zones. If the columns `"zone1"` and `"zone2"` are present, then the argument to `zones` must be `NULL`.

`data` **as an** `array` **object** 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

Williams JC, ReVelle CS, and Levin SA (2005) Spatial attributes and reserve design models: A review. *Environmental Modeling and Assessment*, 10: 163–181.

See Also

Other functions for adding penalties: `add_asym_connectivity_penalties()`, `add_boundary_penalties()`, `add_connectivity_penalties()`, `add_feature_weights()`, `add_linear_penalties()`

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
```

```
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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 low neighbor penalties and
# using a rook-style neighborhood (the default neighborhood style)
p2 <- p1 %>% add_neighbor_penalties(0.001)

# create problem with high penalties
# using a rook-style neighborhood (the default neighborhood style)
p3 <- p1 %>% add_neighbor_penalties(0.01)

# create problem with high penalties and using a queen-style neighborhood
p4 <-
  p1 %>%
  add_neighbor_penalties(
    0.01, data = adjacency_matrix(sim_pu_raster, directions = 8)
  )

# solve problems
s1 <- c(solve(p1), solve(p2), solve(p3), solve(p4))
names(s1) <- c("basic solution", "low (rook)", "high (rook)", "high (queen")

# plot solutions
plot(s1, axes = FALSE)

# create minimal problem with multiple zones
p5 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  add_min_set_objective() %>%
  add_relative_targets(matrix(0.1, ncol = 3, nrow = 5)) %>%
  add_default_solver(verbose = FALSE)

# create problem with low neighbor penalties, a rook style neighborhood,
# and planning units are only considered neighbors if they are allocated to
# the same zone
z6 <- diag(3)
print(z6)
p6 <- p5 %>% add_neighbor_penalties(0.001, zones = z6)

# create problem with high penalties and the same neighborhood as above
p7 <- p5 %>% add_neighbor_penalties(0.01, zones = z6)

# create problem with high neighborhood penalties, a queen-style
# neighborhood, neighboring planning units that are allocated to zones 1
# or 2 are treated as neighbors
z8 <- diag(3)
```

```
z8[1, 2] <- 1
z8[2, 1] <- 1
print(z8)
p8 <- p5 %>% add_neighbor_penalties(0.01, zones = z8)

# create problem with high neighborhood penalties, a queen-style
# neighborhood, and here we want to promote spatial fragmentation
# within each zone, so we use negative zone values.
z9 <- diag(3) * -1
print(z9)
p9 <- p5 %>% add_neighbor_penalties(0.01, zones = z9)

# solve problems
s2 <- list(p5, p6, p7, p8, p9)
s2 <- lapply(s2, solve)
s2 <- lapply(s2, category_layer)
s2 <- terra::rast(s2)
names(s2) <- c("basic problem", "p6", "p7", "p8", "p9")

# plot solutions
plot(s2, main = names(s2), 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()` 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). Only a single decision should be added to a `problem()` object. Note that if multiple decisions are added to an object, then the last one to be added will be used.

Value

An updated problem() object with the decisions added to it.

See Also

See decisions for an overview of all functions for adding decisions.

Other decisions: add_binary_decisions(), add_semicontinuous_decisions()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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)

# solve problem
s1 <- solve(p1)

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

# build multi-zone conservation problem with proportion decisions
p2 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  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)

# 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)
```

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

`add_relative_targets` *Add relative targets*

Description

Add targets to a conservation planning problem expressed as a proportion (between 0 and 1) of the maximum level of representation of each feature 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 values) 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()` object.
`targets`	Object that specifies the targets for each feature. See the Targets format section for more information.

Details

This function is used to set targets for each feature (separately). For problems associated with a single management zone, this function may be useful to specify individual targets for each feature. For problems associated with multiple management zones, this function can also be used to specify a target for each feature within each zone (separately). For example, this may be useful in planning exercises where it is important to ensure that some of the features are adequately represented by multiple zones. For example, in a marine spatial planning exercise, it may be important for some features (e.g., commercial important fish species) to be adequately represented by a conservation zone for ensuring their long-term persistence, and also by a fishing zone to for ensure food security. For greater flexibility in target setting (such as setting targets that can be met through the allocation of multiple zones), see the `add_manual_targets()` function.

Value

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

Target setting

Many conservation planning problems require targets. Targets are used to specify the minimum amount, or proportion, of a feature's spatial distribution that should ideally be protected. This is important so that the optimization process can weigh the merits and trade-offs between improving the representation of one feature over another feature. Although it can be challenging to set meaningful targets, this is a critical step for ensuring that prioritizations meet the stakeholder objectives that underpin a prioritization exercise (Carwardine *et al.* 2009). In other words, targets play an important role in ensuring that a priority setting process is properly tuned according to stakeholder requirements. For example, targets provide a mechanism for ensuring that a prioritization secures enough habitat to promote the long-term persistence of each threatened species, culturally important species, or economically important ecosystem services under consideration. Since there is often uncertainty regarding stakeholder objectives (e.g., how much habitat should be protected for a given species) or the influence of particular target on a prioritization (e.g., how would setting a 90% or 100% for a threatened species alter priorities), it is often useful to generate and compare a suite of prioritizations based on different target scenarios.

Targets format

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

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

`targets` **as a** `matrix` **object** 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.

`targets` **as a** `character` **vector** containing the column name(s) in the feature data associated with the argument to `x` that contain targets. This format can only be used when the feature data associated with `x` is a `sf::st_sf()` or `data.frame`. For problems that contain a single zone, the argument to `targets` must contain a single column name. Otherwise, for problems that contain multiple zones, the argument to `targets` must contain a column name for each zone.

References

Carwardine J, Klein CJ, Wilson KA, Pressey RL, Possingham HP (2009) Hitting the target and missing the point: target-based conservation planning in context. *Conservation Letters*, 2: 4–11.

See Also

Other functions for adding targets: `add_absolute_targets()`, `add_auto_targets()`, `add_group_targets()`, `add_manual_targets()`

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)
```

```
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_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)

# solve problem
s3 <- c(solve(p1), solve(p2))
names(s3) <- c("10% targets", "varying targets")

# plot solution
plot(s3, axes = FALSE)

# create a problem with multiple management zones
p4 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  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_zones_features), zone_names(sim_zones_features)
  )
)
print(p4_targets)

p5 <- p4 %>% add_relative_targets(p4_targets)

# solve problem
s5 <- solve(p5)

# plot solution (cell values correspond to zone identifiers)
plot(category_layer(s5), main = "equal targets")

# create a problem with targets that require a varying amount of each
# feature to be represented in each zone
```

```
p6_targets <- matrix(
  runif(15, 0.01, 0.2), nrow = 5, ncol = 3, dimnames = list(
    feature_names(sim_zones_features), zone_names(sim_zones_features)
  )
)
print(p6_targets)

p6 <- p4 %>% add_relative_targets(p6_targets)

# solve problem
s6 <- solve(p6)

# plot solution (cell values correspond to zone identifiers)
plot(category_layer(s6), main = "varying targets")

## End(Not run)
```

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

Description

Specify that the *SYMPHONY* software – using the **Rsymphony** package – should be used to solve a conservation planning problem (Ralphs & Güzelsoy 2005). 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()` 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

An updated `problem()` object 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

See solvers for an overview of all functions for adding a solver.

Other functions for adding solvers: `add_cbc_solver()`, `add_cplex_solver()`, `add_default_solver()`, `add_gurobi_solver()`, `add_highs_solver()`, `add_lsymphony_solver`

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_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)

## 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. 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 prioritized. 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()` 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). Only a single decision should be added to a `problem()` object. Note that if multiple decisions are added to an object, then the last one to be added will be used.

Value

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

See Also

See decisions for an overview of all functions for adding decisions.

Other decisions: `add_binary_decisions()`, `add_proportion_decisions()`

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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)

# solve problem
s1 <- solve(p1)

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

# build multi-zone conservation problem with semi-continuous decisions
p2 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  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)

# 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)

## 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. Although this function can be useful for generating multiple different solutions for a given problem, it is recommended to use add_pool_portfolio if the *Gurobi* software is available.

Usage

```
add_shuffle_portfolio(
  x,
  number_solutions = 10,
  threads = 1,
  remove_duplicates = TRUE
)
```

Arguments

x	problem() 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

An updated problem() object with the portfolio added to it.

See Also

See portfolios for an overview of all functions for adding a portfolio.

Other functions for adding portfolios: add_cuts_portfolio(), add_default_portfolio(), add_extra_portfolio(), add_gap_portfolio(), add_top_portfolio()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
```

```
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# 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)

# solve problem and generate 10 solutions within 20% of optimality
s1 <- solve(p1)

# convert portfolio into a multi-layer raster
s1 <- terra::rast(s1)

# print number of solutions found
print(terra::nlyr(s1))

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

# build multi-zone conservation problem with shuffle portfolio
p2 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  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)

# solve the problem
s2 <- solve(p2)

# convert each solution in the portfolio into a single category layer
s2 <- terra::rast(lapply(s2, category_layer))

# print number of solutions found
print(terra::nlyr(s2))

# plot solutions in portfolio
plot(s2, axes = 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 = 10)
```

Arguments

x	problem() object.
number_solutions	integer number of solutions required. Defaults to 10.

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. 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

An updated problem() object with the portfolio added to it.

See Also

See portfolios for an overview of all functions for adding a portfolio.

Other functions for adding portfolios: add_cuts_portfolio(), add_default_portfolio(), add_extra_portfolio(), add_gap_portfolio(), add_shuffle_portfolio()

Examples

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

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_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)

# convert portfolio into a multi-layer raster
s1 <- terra::rast(s1)

# print number of solutions found
print(terra::nlyr(s1))

# plot solutions
plot(s1, axes = FALSE)

# create multi-zone problem with a portfolio for the top 5 solutions
p2 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  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)

# convert each solution in the portfolio into a single category layer
s2 <- terra::rast(lapply(s2, category_layer))

# print number of solutions found
print(terra::nlyr(s2))

# plot solutions in portfolio
plot(s2, axes = FALSE)

## End(Not run)
```

`adjacency_matrix` *Adjacency matrix*

Description

Create a matrix showing which planning units are spatially adjacent to each other.

Usage

```
adjacency_matrix(x, ...)

## S3 method for class 'Raster'
adjacency_matrix(x, directions = 4, ...)
```

```
## S3 method for class 'SpatRaster'
adjacency_matrix(x, directions = 4, ...)

## 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	terra::rast() or sf::sf() object representing planning units.
...	not used.
directions	integer If x is a terra::rast() 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 sf::sf() objects, and terra::adjacent() for terra::rast() objects. Note that spatially overlapping planning units are considered adjacent.

Value

A Matrix::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 terra::rast() 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

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_pu_polygons <- get_sim_pu_polygons()
```

```
# create adjacency matrix using raster data
## crop raster to 9 cells
r <- terra::crop(sim_pu_raster, terra::ext(c(0, 0.3, 0, 0.3)))

## make adjacency matrix
am_raster <- adjacency_matrix(r)

# create adjacency matrix using polygon data
## subset 9 polygons
ply <- sim_pu_polygons[c(1:3, 11:13, 20:22), ]

## make adjacency matrix
am_ply <- adjacency_matrix(ply)

# plot data and the adjacency matrices

## plot raster and adjacency matrix
plot(r, main = "raster", axes = FALSE)
Matrix::image(am_raster, main = "adjacency matrix")

## plot polygons and adjacency matrix
plot(ply[, 1], main = "polygons")
Matrix::image(am_ply, main = "adjacency matrix")


## End(Not run)
```

---

as_km2 *Standardize unit to km^2*

Description

Standardize number to km^2.

Usage

```
as_km2(x, unit)
```

Arguments

x	numeric vector.
unit	character vector of spatial units (e.g., "km2", "acres", "hectares". For convenience, a single character value can be specified if all values in x are the same unit.

Value

A numeric vector.

Examples

```
as_km2(5, "km2")
as_km2(5, "acres")
as_km2(c(5, 10), "ha")
as_km2(c(5, 10), c("ha", "acres"))
```

`as_per_km2` *Standardize unit to density per km^2*

Description

Standardize number to density per km^2.

Usage

```
as_per_km2(x, unit)
```

Arguments

`x`	`numeric` vector.
`unit`	`character` vector of spatial units (e.g., `"km2"`, `"acres"`, `"hectares"`. For convenience, a single `character` value can be specified if all values in `x` are the same unit.

Value

A `numeric` vector.

Examples

```
as_per_km2(5, "km2")
as_per_km2(5, "acres")
as_per_km2(c(5, 10), "ha")
as_km2(c(5, 10), c("ha", "acres"))
```

`binary_stack` *Binary stack*

Description

Convert a single-layer `terra::rast()` object that contains integer values into a multi-layer `terra::rast()` object with cell values denote the presence/absence of a given integer value. This is methodology is also known as "one-hot encoding".

Usage

```
binary_stack(x, keep_all = TRUE)

## S3 method for class 'Raster'
binary_stack(x, keep_all = TRUE)

## S3 method for class 'SpatRaster'
binary_stack(x, keep_all = TRUE)
```

Arguments

`x`	`terra::rast()` object with a single layer that contains integer values.
`keep_all`	`logical` value indicating if all integers should be kept in the output. If `TRUE`, the output will contain a layer for each sequential integer between 1 and the maximum value in `x`. If `FALSE`, the output will only contain layers for integer values present in `x`. Defaults to `TRUE`.

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 rasters as input arguments. It is essentially a wrapper for `terra::segregate()`. Note that this function assumes `x` contains integer values.

Value

A `terra::rast()` object.

See Also

The `category_layer()` function performs the reverse of this function. Also the `terra::segregate()` function provides similar functionality.

Examples

```
# create raster with categorical values
x <- terra::rast(matrix(c(1, 2, 4, 0, NA, 1), nrow = 3))

# plot the raster
## Not run:
plot(x, main = "x")

## End(Not run)

# convert to binary stack
y <- binary_stack(x)

# plot result
## Not run:
plot(y)

## End(Not run)
```

---

boundary_matrix *Boundary matrix*

Description

Generate a matrix describing the amount of shared boundary length between different planning units, and the total amount of boundary length for each planning unit.

Usage

```
boundary_matrix(x, ...)

## S3 method for class 'Raster'
boundary_matrix(x, ...)

## S3 method for class 'SpatRaster'
boundary_matrix(x, ...)

## S3 method for class 'SpatialPolygons'
boundary_matrix(x, ...)

## S3 method for class 'SpatialLines'
boundary_matrix(x, ...)

## S3 method for class 'SpatialPoints'
boundary_matrix(x, ...)

## S3 method for class 'sf'
boundary_matrix(x, ...)
```

```
## Default S3 method:
boundary_matrix(x, ...)
```

Arguments

x	terra::rast() or sf::sf() object representing planning units.
...	not used.

Details

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 terra::rast() objects boundaries are missing for cells that have missing (NA) values in all cells.

Value

A Matrix::dsCMatrix symmetric sparse matrix object. Each row and column represents a planning unit. Cell values indicate the shared boundary length between different pairs of planning units. Values along the matrix diagonal indicate the total perimeter associated with each planning unit.

Notes

In earlier versions, this function had an extra str_tree parameter that could be used to leverage STR query trees to speed up processing for planning units in vector format. Although this functionality improved performance, it was not enabled by default because the underlying function (i.e., rgeos:gUnarySTRtreeQuery()) was documented as experimental. The boundary_matrix() function has since been updated so that it will use STR query trees to speed up processing for planning units in vector format (using terra::sharedPaths()).

Also, note that in previous versions, cell values along the matrix diagonal indicated the perimeter associated with planning units that did not contain any neighbors. This has now changed such that values along the diagonal now correspond to the total perimeter associated with each planning unit.

See Also

Boundary matrix data might need rescaling to improve optimization performance, see rescale_matrix() to perform these calculations.

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_pu_polygons <- get_sim_pu_polygons()

# subset data to reduce processing time
r <- terra::crop(sim_pu_raster, c(0, 0.3, 0, 0.3))
ply <- sim_pu_polygons[c(1:3, 11:13, 20:22), ]
```

```
# create boundary matrix using raster data
bm_raster <- boundary_matrix(r)

# create boundary matrix using polygon data
bm_ply <- boundary_matrix(ply)

# plot raster data
plot(r, main = "raster", axes = FALSE)

# plot boundary matrix
# here each row and column corresponds to a different planning unit
Matrix::image(bm_raster, main = "boundary matrix")

# plot polygon data
plot(ply[, 1], main = "polygons", axes = FALSE)

# plot boundary matrix
# here each row and column corresponds to a different planning unit
Matrix::image(bm_ply, main = "boundary matrix")

## 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

A Matrix::dgCMatrix sparse matrix object. Each row corresponds to a different species. Each column corresponds to a different branch. Species that inherit from a given branch are denoted with a one.

Examples

```
# load data
sim_phylogeny <- get_sim_phylogeny()

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

# plot data
## Not run:
plot(sim_phylogeny, main = "phylogeny")
Matrix::image(m, main = "branch matrix")

## End(Not run)
```

---

calibrate_cohon_penalty

Calibrate penalties with Cohon's method

Description

Identify a penalty value that represents a suitable compromise between the primary objective and a penalty for a conservation planning problem. This is accomplished following the multi-objective algorithm developed by Cohon *et al.* (1979) that was later adapted for systematic conservation planning (Ardron *et al.* 2010; Fischer and Church 2005).

Usage

```
calibrate_cohon_penalty(x, approx = TRUE, verbose = TRUE)
```

Arguments

x	problem() object with a penalty.
approx	logical value indicating if an approximation method should be used for the calculations. Defaults to TRUE to reduce computational burden. See Details section for more information.
verbose	logical should information be printed while solving optimization problems? Defaults to TRUE.

Details

This function provides a routine for implementing Cohon's method (1979) to identify a suitable penalty value. It can be used calibrate a broad range of penalties, including boundary penalties (`add_boundary_penalties()`), connectivity penalties (`add_connectivity_penalties()`), asymmetric connectivity penalties (`add_asym_connectivity_penalties()`), and linear penalties (`add_linear_penalties()`). Note that the penalty value identified by this function is calculated in a manner that reflects the overall problem formulation (per `x`). Thus if you are considering multiple scenarios that consider different objectives, constraints, penalties, targets, decision types, or underlying datasets, then you will likely need to re-run the calibration process to identify a suitable penalty value for each scenario (separately).

The suitability of the resulting penalty value depends on the optimality gap used during optimization, as well as whether the approximation method is used or not. In particular, a gap of zero will result in the best estimate, and a gap greater than zero may result in worse estimates. It is recommended to keep the optimality gap low (e.g., between 0 and 0.1), and a relatively small gap may be needed in some cases (e.g., 0, 0.01 or 0.05). Additionally, the approximation method (i.e., with `approx = TRUE`) may result in penalty values that do not represent a suitable compromise between the objective and the penalty. Although this will happen if there are multiple optimal solutions to the primary objective or the penalty; in practice, this is unlikely to be an issue when considering a moderate number of features and planning units. If this is an issue, then a more robust approach can be employed (i.e., by setting `approx = FALSE`) that uses additional optimization routines to potentially obtain a better estimate of a suitable penalty value. As such, it is recommended to try running the function with default settings and see if the resulting penalty value is suitable. If not, then try running it with a smaller optimality gap or the robust approach.

Value

A `numeric` value corresponding to the calibrated penalty value. Additionally, this value has attributes that contain the values used to calculate the calibrated penalty value. These attributes include the (`solution_1_objective`) optimal objective value, (`solution_1_penalty`) best possible penalty value given that the solution must be optimal according to the primary objective, (`solution_2_penalty`) optimal penalty value, and (`solution_2_objective`) best possible objective value given that a solution must be optimal according to the penalties.

Mathematical formulation

A suitable penalty value is identified using the following procedure.

1. The optimal value for the primary objective is calculated (referred to as `solution_1_objective` in the output). This is accomplished by solving the problem without the penalty. For example, if considering a minimum set problem with boundary penalties, then this value would correspond to the cost of the solution that has the smallest cost (whilst meeting all the targets).
2. The best possible penalty value given that the solution must be optimal according to the primary objective is calculated (referred to as `solution_1_penalty` in the output). For example, if considering a minimum set problem with boundary penalties, then this value would correspond to the smallest possible total boundary length that is possible when a solution must have minimum cost (whilst meeting all the targets). If using the approximation method (per `approx = TRUE`), this value is estimated based on the penalty value of the solution produced in the previous step. Otherwise, if using the robust method (per `approx = FALSE`), this value is

calculated by performing an additional optimization routine to ensure that this value is correct when there are multiple optimal solutions.

3. The optimal value for the penalty is calculated (referred to as `solution_2_penalty` in the output). This is accomplished by modifying the problem so that it only focuses on minimizing the penalty as much as possible (i.e., ignoring the primary objective) and solving it. For example, if considering a minimum set problem with boundary penalties, then this value would correspond to the total boundary length of the solution that has the smallest total boundary length (while still meeting all the targets).
4. The best possible objective value given that the solution must be optimal according to the penalties is calculated (referred to as `solution_2_objective` in the output). For example, if considering a minimum set problem with boundary penalties, then this value would correspond to the smallest possible cost that is possible when a solution must have the minimum boundary length (whilst meeting all the targets). If using the approximation method (per `approx = TRUE`), this value is estimated based on the objective value of the solution produced in the previous step. Otherwise, if using the robust method (per `approx = FALSE`), this value is calculated by performing an additional optimization routine to ensure that this value is correct when there are multiple optimal solutions.
5. After completing the previous calculations, the suitable penalty value is calculated using the following equation. Here, the values calculated in steps 1, 2, 3, and 4 correspond to a, b, c, and d (respectively).

$$\left|\frac{(a-c)}{(b-d)}\right|$$

References

Ardron JA, Possingham HP, and Klein CJ (eds) (2010) Marxan Good Practices Handbook, Version 2. Pacific Marine Analysis and Research Association, Victoria, BC, Canada.

Cohon JL, Church RL, and Sheer DP (1979) Generating multiobjective trade-offs: An algorithm for bicriterion problems. *Water Resources Research*, 15: 1001–1010.

Fischer DT and Church RL (2005) The SITES reserve selection system: A critical review. *Environmental Modeling and Assessment*, 10: 215–228.

See Also

See penalties for an overview of all functions for adding penalties.

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()

# create problem with boundary penalties
## note that we use penalty = 1 as a place-holder
p1 <-
```

```
  problem(sim_pu_raster, sim_features) %>%
  add_min_set_objective() %>%
  add_boundary_penalties(penalty = 1) %>%
  add_relative_targets(0.2) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# find calibrated boundary penalty using Cohon's method
cohon_penalty <- calibrate_cohon_penalty(p1, verbose = FALSE)

# create a new problem with the calibrated boundary penalty
p2 <-
  problem(sim_pu_raster, sim_features) %>%
  add_min_set_objective() %>%
  add_boundary_penalties(penalty = cohon_penalty) %>%
  add_relative_targets(0.2) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s2 <- solve(p2)

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

## End(Not run)
```

`category_layer` *Category layer*

Description

Convert a multi-layer `terra::rast()` object into a single-layer `terra::rast()` object where pixel values indicate which input layer had the greatest value.

Usage

```
category_layer(x)

## S3 method for class 'Raster'
category_layer(x)

## Default S3 method:
category_layer(x)
```

Arguments

x	`terra::rast()` object containing multiple layers.

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. It is essentially a wrapper for `terra::which.max()`.

Value

A `terra::rast()` object.

See Also

The `binary_stack()` function performs the reverse of this function.

Examples

```
# create a binary raster stack
x <- terra::rast(list(
 terra::rast(matrix(c(1, 0, 0, 1, NA, 0), nrow = 3)),
 terra::rast(matrix(c(0, 1, 0, 0, NA, 0), nrow = 3)),
 terra::rast(matrix(c(0, 0, 1, 0, NA, 1), nrow = 3))
))

# plot data
plot(x)

# convert to category layer
y <- category_layer(x)

# plot result
## Not run:
plot(y)

## End(Not run)
```

`category_vector` *Category vector*

Description

Convert an object containing binary (`integer`) columns into a `integer` vector indicating the column index where each row has the highest value.

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, or sf::sf() object.

Details

This function is conceptually similar to base::max.col() except that rows with all values equal to zero value are assigned a value of zero.

Value

An integer vector.

See Also

The base::max.col() provides similar functionality.

Examples

```
# create matrix with logical columns
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)
```

---

compile *Compile a problem*

Description

Compile a conservation planning problem into an mixed integer linear programming problem.

Usage

```
compile(x, ...)

## S3 method for class 'ConservationProblem'
compile(x, compressed_formulation = NA, ...)
```

Arguments

x	problem() 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 `compressed_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

A optimization_problem() object.

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()

# 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)

## End(Not run)
```

`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 'sf,SpatRaster'
connectivity_matrix(x, y, ...)

## S4 method for signature 'Raster,Raster'
connectivity_matrix(x, y, ...)

## S4 method for signature 'SpatRaster,SpatRaster'
connectivity_matrix(x, y, ...)
```

Arguments

x	`terra::rast()` or `sf::sf()` object representing planning units.
y	`terra::rast()` 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 `sf::sf()` object. Also, note that if the argument to `x` is a `terra::rast()` 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 sf::sf() object and argument to y is a terra::rast() object.

Details

Shared boundary calculations are performed using boundary_matrix().

Value

A Matrix::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 terra::rast() object, then cells with missing (NA) values are set to zero too.

See Also

Connectivity matrix data might need rescaling to improve optimization performance, see rescale_matrix() to perform these calculations.

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()

# create connectivity matrix using raster planning unit data using
# the raster cost values to represent conductance
## extract 9 planning units
r <- terra::crop(sim_pu_raster, terra::ext(c(0, 0.3, 0, 0.3)))

## extract conductance data for the 9 planning units
cd <- terra::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
plot(r, main = "planning units (raster)", axes = FALSE)
plot(cd[[2]], main = "conductivity", axes = FALSE)
Matrix::image(cm_raster,  main = "connectivity matrix")

# 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_polygons[c(1:3, 11:13, 20:22), ]
```

```
## make connectivity matrix
cm_ply <- connectivity_matrix(ply, sim_features[[2]])

## plot data and matrix
plot(sf::st_geometry(ply), main = "planning units (polygons)")
plot(terra::crop(sim_features[[2]], ply), main = "connectivity")
Matrix::image(cm_ply, main = "connectivity matrix")

# create connectivity matrix using habitat suitability data for each feature,
# this could be useful if prioritizations 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) 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
plot(r, main = "planning units (raster)", axes = FALSE)
Matrix::image(cm_sum, main = "connectivity matrix")

## 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
plot(r, main = "planning units (raster)", axes = FALSE)
Matrix::image(cm_wsum, main = "connectivity matrix")

## 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

### 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, max = 1) # rescale matrices
```

```
cm_lwsum <- Map("*", cm_lwsum, weights) # multiply by weights
cm_lwsum <- Reduce("+", cm_lwsum) # sum matrices together

## plot data and matrix
plot(r, main = "planning units (raster)", axes = FALSE)
Matrix::image(cm_lwsum, main = "connectivity matrix")

## 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
plot(r, main = "planning units (raster)", axes = FALSE)
Matrix::image(cm_zwsum, main = "connectivity matrix")

## End(Not run)
```

`ConservationModifier-class`

Conservation problem modifier class

Description

This super-class is used to construct `Objective` `Penalty`, `Target`, `Constraint`, `Portfolio`, `Solver`, and `Decision` objects. **Only experts should use the fields and methods for this class directly.**

Public fields

`name` `character` value.

`data` `list` containing data.

`internal` `list` containing internal computed values.

`compressed_formulation` `logical` value indicating if the object is compatible with a compressed formulation.

Methods

Public methods:

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

Method `print()`**:** Print information about the object.

Usage:

```
ConservationModifier$print()
```

Returns: None.

Method `show()`**:** Print information about the object.

Usage:

```
ConservationModifier$show()
```

Returns: None.

Method `repr()`**:** Generate a character representation of the object.

Usage:

```
ConservationModifier$repr(compact = TRUE)
```

Arguments:

`compact` `logical` value indicating if the output value should be compact? Defaults to `FALSE`.

Returns: A `character` value.

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

Usage:

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

Arguments:

`x` `optimization_problem()` object.

`y` `problem()` object.

Returns: Invisible `TRUE`.

Method `get_data()`**:** Get values stored in the `data` field.

Usage:

```
ConservationModifier$get_data(x)
```

Arguments:

`x` `character` name of data.

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

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

Usage:

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

Arguments:

`x` `character` name of data.

`value` Object to store.

Returns: Invisible `TRUE`.

Method `get_internal()`**:** Get values stored in the `internal` field.

Usage:

```
ConservationModifier$get_internal(x)
```

Arguments:

`x` `character` name of data.

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

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

Usage:

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

Arguments:

`x` `character` name of data.

`value` Object to store.

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

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other classes: `ConservationProblem-class`, `Constraint-class`, `Decision-class`, `Objective-class`, `OptimizationProblem-class`, `Penalty-class`, `Portfolio-class`, `Solver-class`, `Target-class`, `TargetMethod-class`, `Weight-class`

ConservationProblem-class
Conservation problem class

Description

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

Public fields

`data` `list` containing data (e.g., planning units, costs).

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

`objective` `Objective` object specifying the objective function for the problem formulation.

`decisions` `Decision` object specifying the decision types for the problem formulation.

`targets` `Target` object specifying the representation targets for the problem formulation.

`weights` `Weight` object specifying the feature weights for the problem formulation.

`constraints` `list` containing `Constraint` objects that specify constraints for the problem formulation.

`penalties` `list` containing `Penalty` objects that specify penalties for the problem formulation.

`portfolio` `Portfolio` object specifying the approach for generating multiple solutions.

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

Methods

Public methods:

- `ConservationProblem$new()`
- `ConservationProblem$summary()`
- `ConservationProblem$print()`
- `ConservationProblem$show()`
- `ConservationProblem$repr()`
- `ConservationProblem$get_data()`
- `ConservationProblem$set_data()`
- `ConservationProblem$number_of_planning_units()`
- `ConservationProblem$is_ids_equivalent_to_indices()`
- `ConservationProblem$planning_unit_indices()`
- `ConservationProblem$total_unit_ids()`
- `ConservationProblem$convert_total_unit_ids_to_indices()`

- ConservationProblem$planning_unit_indices_with_finite_costs()
- ConservationProblem$set_planning_unit_indices_with_finite_costs()
- ConservationProblem$number_of_total_units()
- ConservationProblem$planning_unit_costs()
- ConservationProblem$planning_unit_class()
- ConservationProblem$set_planning_unit_costs()
- ConservationProblem$number_of_features()
- ConservationProblem$feature_names()
- ConservationProblem$feature_abundances_in_planning_units()
- ConservationProblem$set_feature_abundances_in_planning_units()
- ConservationProblem$feature_positive_abundances_in_planning_units()
- ConservationProblem$set_feature_positive_abundances_in_planning_units()
- ConservationProblem$feature_abundances_in_total_units()
- ConservationProblem$feature_units()
- ConservationProblem$feature_abundances_km2_in_total_units()
- ConservationProblem$set_feature_abundances_km2_in_total_units()
- ConservationProblem$feature_targets()
- ConservationProblem$feature_weights()
- ConservationProblem$has_negative_feature_data()
- ConservationProblem$number_of_zones()
- ConservationProblem$zone_names()
- ConservationProblem$add_portfolio()
- ConservationProblem$add_solver()
- ConservationProblem$add_targets()
- ConservationProblem$add_weights()
- ConservationProblem$add_objective()
- ConservationProblem$add_decisions()
- ConservationProblem$add_constraint()
- ConservationProblem$add_penalty()
- ConservationProblem$remove_all_penalties()
- ConservationProblem$clone()

Method `new()`**:** Create a new conservation problem object.

Usage:

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

Arguments:

`data` `list` containing data

Returns: A new `ConservationProblem` object.

Method `summary()`**:** Print extended information about the object.

Usage:

```
ConservationProblem$summary()
```

Returns: Invisible TRUE.

Method print(): Print concise information about the object.

Usage:

```
ConservationProblem$print()
```

Returns: Invisible TRUE.

Method show(): Display concise information about the object.

Usage:

```
ConservationProblem$show()
```

Returns: Invisible TRUE.

Method repr(): Generate a character representation of the object.

Usage:

```
ConservationProblem$repr()
```

Returns: A character value.

Method get_data(): Get values stored in the data field.

Usage:

```
ConservationProblem$get_data(x)
```

Arguments:

x character name of data.

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

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

Usage:

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

Arguments:

x character name of data.

value Object to store.

Returns: Invisible TRUE.

Method number_of_planning_units(): Obtain the number of planning units. The planning units correspond to elements in the cost data (e.g., indices, rows, geometries, cells) that have finite values in at least one zone. In other words, planning unit are elements in the cost data that do not have missing (NA) values in every zone.

Usage:

```
ConservationProblem$number_of_planning_units()
```

Returns: An integer value.

Method is_ids_equivalent_to_indices(): Check if planning unit identifiers are equivalent to the planning unit indices? Only FALSE if the planning units are data.frame format.

Usage:

```
ConservationProblem$is_ids_equivalent_to_indices()
```

Returns: A `logical` value.

Method `planning_unit_indices()`**:** Obtain the planning unit indices.

Usage:

```
ConservationProblem$planning_unit_indices()
```

Returns: An `integer` vector.

Method `total_unit_ids()`**:** Obtain the total unit identifiers.

Usage:

```
ConservationProblem$total_unit_ids()
```

Returns: An `integer` vector.

Method `convert_total_unit_ids_to_indices()`**:** Convert total unit identifiers to indices.

Usage:

```
ConservationProblem$convert_total_unit_ids_to_indices(ids)
```

Arguments:

`ids` `integer` vector with planning unit identifiers.

Returns: An `integer` vector.

Method `planning_unit_indices_with_finite_costs()`**:** Obtain the planning unit indices that are associated with finite cost values.

Usage:

```
ConservationProblem$planning_unit_indices_with_finite_costs()
```

Returns: A `list` of `integer` vectors. Each `list` element corresponds to a different zone.

Method `set_planning_unit_indices_with_finite_costs()`**:** Perform calculations to cache the planning unit indices that are associated with finite cost values.

Usage:

```
ConservationProblem$set_planning_unit_indices_with_finite_costs()
```

Returns: Invisible `TRUE`.

Method `number_of_total_units()`**:** Obtain the number of total units. The total units include all elements in the cost data (e.g., indices, rows, geometries, cells), including those with missing (`NA`) values.

Usage:

```
ConservationProblem$number_of_total_units()
```

Returns: An `integer` value.

Method `planning_unit_costs()`**:** Obtain the planning unit costs.

Usage:

```
ConservationProblem$planning_unit_costs()
```

Returns: A numeric matrix.

Method planning_unit_class(): Get planning unit class.

Usage:

```
ConservationProblem$planning_unit_class()
```

Returns: A character value.

Method set_planning_unit_costs(): Perform calculations to cache the planning unit costs.

Usage:

```
ConservationProblem$set_planning_unit_costs()
```

Returns: Invisible TRUE.

Method number_of_features(): Obtain the number of features.

Usage:

```
ConservationProblem$number_of_features()
```

Returns: An integer value.

Method feature_names(): Obtain the names of the features.

Usage:

```
ConservationProblem$feature_names()
```

Returns: A character vector.

Method feature_abundances_in_planning_units(): Obtain the abundance of the features in the planning units.

Usage:

```
ConservationProblem$feature_abundances_in_planning_units()
```

Returns: A numeric matrix. Each column corresponds to a different zone and each row corresponds to a different feature.

Method set_feature_abundances_in_planning_units(): Perform calculations to cache the abundance of the features in the planning units.

Usage:

```
ConservationProblem$set_feature_abundances_in_planning_units()
```

Returns: Invisible TRUE.

Method feature_positive_abundances_in_planning_units(): Obtain the positive abundance of the features in the planning units. Note that this method, unlike feature_abundances_in_planning_units,

Usage:

```
ConservationProblem$feature_positive_abundances_in_planning_units()
```

Returns: A numeric matrix. Each column corresponds to a different zone and each row corresponds to a different feature.

Method set_feature_positive_abundances_in_planning_units(): Perform calculations to cache the positive abundance of the features in the planning units.

Usage:

```
ConservationProblem$set_feature_positive_abundances_in_planning_units()
```

Returns: Invisible `TRUE`.

Method `feature_abundances_in_total_units()`**:** Obtain the abundance of the features in the total units.

Usage:

```
ConservationProblem$feature_abundances_in_total_units()
```

Returns: A `numeric` matrix. Each column corresponds to a different zone and each row corresponds to a different feature.

Method `feature_units()`**:** Obtain the units of the features.

Usage:

```
ConservationProblem$feature_units()
```

Returns: A `character` value. Each element corresponds to a different feature.

Method `feature_abundances_km2_in_total_units()`**:** Obtain the abundance of the features in area-based units of km^2.

Usage:

```
ConservationProblem$feature_abundances_km2_in_total_units()
```

Details: Note that if a feature has missing (`NA`) units then missing values are returned.

Returns: A `numeric` matrix. Each column corresponds to a different zone and each row corresponds to a different feature.

Method `set_feature_abundances_km2_in_total_units()`**:** Perform calculations to cache the abundance of the features in area-based units of km^2.

Usage:

```
ConservationProblem$set_feature_abundances_km2_in_total_units()
```

Returns: Invisible `TRUE`.

Method `feature_targets()`**:** Obtain the representation targets for the features.

Usage:

```
ConservationProblem$feature_targets()
```

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

Method `feature_weights()`**:** Obtain the weights for the features.

Usage:

```
ConservationProblem$feature_weights()
```

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

Method `has_negative_feature_data()`**:** See if the feature data contain any negative values.

Usage:

```
ConservationProblem$has_negative_feature_data()
```

Returns: A `logical` value.

Method `number_of_zones()`**:** Obtain the number of zones.

Usage:

```
ConservationProblem$number_of_zones()
```

Returns: An `integer` value.

Method `zone_names()`**:** Obtain the zone names.

Usage:

```
ConservationProblem$zone_names()
```

Returns: A `character` vector.

Method `add_portfolio()`**:** Create a new object with a portfolio added to the problem formulation.

Usage:

```
ConservationProblem$add_portfolio(x)
```

Arguments:

`x` Portfolio object.

Returns: An updated `ConservationProblem` object.

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

Usage:

```
ConservationProblem$add_solver(x)
```

Arguments:

`x` Solver object.

Returns: An updated `ConservationProblem` object.

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

Usage:

```
ConservationProblem$add_targets(x)
```

Arguments:

`x` Target object.

Returns: An updated `ConservationProblem` object.

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

Usage:

```
ConservationProblem$add_weights(x)
```

Arguments:

`x` Weight object.

Returns: An updated `ConservationProblem` object.

Method `add_objective()`**:** Create a new object with an objective added to the problem formulation.

Usage:

```
ConservationProblem$add_objective(x)
```

Arguments:

`x` Objective object.

Returns: An updated `ConservationProblem` object.

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

Usage:

```
ConservationProblem$add_decisions(x)
```

Arguments:

`x` Decision object.

Returns: An updated `ConservationProblem` object.

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

Usage:

```
ConservationProblem$add_constraint(x)
```

Arguments:

`x` Constraint object.

Returns: An updated `ConservationProblem` object.

Method `add_penalty()`**:** Create a new object with a penalty added to the problem formulation.

Usage:

```
ConservationProblem$add_penalty(x)
```

Arguments:

`x` Penalty object.

Returns: An updated `ConservationProblem` object.

Method `remove_all_penalties()`**:** Create a new object without any penalties.

Usage:

```
ConservationProblem$remove_all_penalties(retain = NULL)
```

Arguments:

`retain` `character` vector of classes to retain. Defaults to `NULL` such that all penalties are excluded.

Returns: An updated `ConservationProblem` object.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other classes: ConservationModifier-class, Constraint-class, Decision-class, Objective-class, OptimizationProblem-class, Penalty-class, Portfolio-class, Solver-class, Target-class, TargetMethod-class, Weight-class

---

Constraint-class *Constraint class*

Description

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

Super class

prioritizr::ConservationModifier -> Constraint

Methods

Public methods:

- Constraint$apply()
- Constraint$clone()

Method apply(): Update an optimization problem formulation.

Usage:

```
Constraint$apply(x)
```

Arguments:

x optimization_problem() object.

Returns: Invisible TRUE.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

Other classes: ConservationModifier-class, ConservationProblem-class, Decision-class, Objective-class, OptimizationProblem-class, Penalty-class, Portfolio-class, Solver-class, Target-class, TargetMethod-class, Weight-class

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 are used to impose rules to ensure that solutions exhibit (or do not exhibit) a particular characteristic. 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 forbidding solutions that do not exhibit a specific characteristic, whereas penalty functions work by than penalizing solutions which do not meet a specific characteristic.

The following functions can be used to add constraints 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. These constraints may be especially useful for reducing spatial fragmentation in large-scale planning problems or when using open source solvers.

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_linear_constraints() Add constraints to ensure that all selected planning units meet certain criteria. For example, they can be used to add multiple budgets, or limit the number of planning units selected in different administrative areas within a study region (e.g., different countries).

add_mandatory_allocation_constraints() 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.

See Also

Other overviews: decisions, importance, objectives, penalties, portfolios, solvers, summaries, targets

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_locked_in_raster <- get_sim_locked_in_raster()
sim_locked_out_raster <- get_sim_locked_in_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()

# solve problems
s <- terra::rast(lapply(list(p1, p2, p3, p4, p5, p6), solve))
names(s) <- c(
  "minimal problem", "locked in", "locked out",
  "neighbor", "contiguity", "feature contiguity"
)

# plot solutions
plot(s, axes = FALSE, nr = 2)

## End(Not run)
```

Decision-class *Decision class*

Description

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

Super class

prioritizr::ConservationModifier -> Decision

Methods

Public methods:

- Decision$apply()
- Decision$clone()

Method apply(): Update an optimization problem formulation.

Usage:

```
Decision$apply(x)
```

Arguments:

x optimization_problem() object.

Returns: Invisible TRUE.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

Other classes: ConservationModifier-class, ConservationProblem-class, Constraint-class, Objective-class, OptimizationProblem-class, Penalty-class, Portfolio-class, Solver-class, Target-class, TargetMethod-class, Weight-class

---

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

The following functions can be used to add a decision type to a conservation planning `problem()`. Note that if multiple of these functions are added to a `problem()`, then only the last function added will be used.

`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

Other overviews: `constraints`, `importance`, `objectives`, `penalties`, `portfolios`, `solvers`, `summaries`, `targets`

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_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)

# solve problem
s <- c(solve(p1), solve(p2), solve(p3), solve(p4))
```

```
names(s) <- c(
  "default (binary)", "binary", "proportion", "semicontinuous (upper = 0.5)"
)
# plot solutions
plot(s)

## End(Not run)
```

`eval_asym_connectivity_summary` *Evaluate asymmetric connectivity of solution*

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. It is specifically designed for asymmetric connectivity data.

Usage

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

## S4 method for signature 'ConservationProblem,ANY,ANY,Matrix'
eval_asym_connectivity_summary(x, solution, zones, data)

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

## S4 method for signature 'ConservationProblem,ANY,ANY,dgCMatrix'
eval_asym_connectivity_summary(x, solution, zones, data)

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

Arguments

x	problem() object.
solution	numeric, matrix, data.frame, terra::rast(), 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 working with multiple zones and the argument to data is a matrix or Matrix object. If the argument to data is an array or data.frame with data for multiple zones (e.g., using the "zone1" and "zone2" column names), 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 the Connectivity metric reported by the *Marxan* software (Ball *et al.* 2009). It is calculated using the same equations used to penalize solutions with asymmetric connectivity data (i.e., add_asym_connectivity_penalties()). Specifically, it is calculated as the sum of the connectivity values (in the argument to data) that correspond pairs of planning units, wherein one planning unit is selected by the solution and the other planning unit is not selected by solution.

Value

A 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).

asym_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** terra::rast() **planning units** The argument to solution be a terra::rast() object where different cells 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** 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.

data **as a** matrix/Matrix **object** 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 **as a** data.frame **object** containing columns that are named "id1", "id2", and "boundary". Here, each row denotes the connectivity between a pair of planning units (per values in the "id1" and "id2" columns) following the *Marxan* format. If the argument to x contains multiple zones, then the "zone1" and "zone2" columns can optionally be provided to manually specify the connectivity values between planning units when they are allocated to specific zones. If the "zone1" and "zone2" columns are present, then the argument to zones must be NULL.

data **as an** array **object** 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

See summaries for an overview of all functions for summarizing solutions. Also, see `add_asym_connectivity_penalties()` to penalize solutions with low asymmetric connectivity.

Other functions for summarizing solutions: `eval_boundary_summary()`, `eval_connectivity_summary()`, `eval_cost_summary()`, `eval_feature_representation_summary()`, `eval_n_summary()`, `eval_target_coverage_summ`

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()
sim_zones_pu_polygons <- get_sim_zones_pu_polygons()
sim_zones_features <- get_sim_zones_features()

# build minimal conservation problem with polygon 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)

# solve the problem
s1 <- solve(p1)

# print solution
print(s1)

# plot solution
plot(s1[, "solution_1"])

# simulate connectivity matrix
# here, we will generate connectivity values randomly
# between all pairs of planning units
```

```
acm1 <- matrix(
  runif(nrow(sim_pu_polygons) ^ 2),
  nrow = nrow(sim_pu_polygons)
)

# calculate connectivity associated with the solution
r1 <- eval_asym_connectivity_summary(p1, s1[, "solution_1"], data = acm1)
print(r1)

# build multi-zone conservation problem with polygon data
p2 <-
  problem(
    sim_zones_pu_polygons, sim_zones_features,
     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
s2 <- solve(p2)

# print solution
print(s2)

# create new column representing the zone id that each planning unit
# was allocated to in the solution
s2$solution <- category_vector(
  s2[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")]
)
s2$solution <- factor(s2$solution)

# plot solution
plot(s2[, "solution"])

# simulate asymmetric connectivity matrix
acm2 <- matrix(
  runif(nrow(sim_zones_pu_polygons) ^ 2),
  nrow = nrow(sim_zones_pu_polygons)
)

# calculate connectivity associated with the solution
r2 <- eval_asym_connectivity_summary(
  p2,
  s2[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")],
  data = acm2
)
print(r2)


## End(Not run)
```

`eval_boundary_summary` *Evaluate boundary length of solution*

Description

Calculate the exposed boundary length (i.e., 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,
  solution,
  edge_factor = rep(0.5, number_of_zones(x)),
  zones = diag(number_of_zones(x)),
  data = NULL
)
```

Arguments

x	`problem()` object.
solution	`numeric`, `matrix`, `data.frame`, `terra::rast()`, 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

A `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.

`data` **as a** `NULL` **value** indicating that the data should be 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).

`data` **as a** `matrix`/`Matrix` **object** 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 denote the total boundary length associated with each planning unit.

`data` **as a** `data.frame` **object** 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. Additionally, if the values in the `"id1"` and `"id2"` columns contain the same values, then the value denotes the amount of exposed boundary length (not total boundary). 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** terra::rast() **planning units** The argument to solution be a terra::rast() object where different cells 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** 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

See summaries for an overview of all functions for summarizing solutions. Also, see add_boundary_penalties() to penalize solutions with high boundary length.

Other functions for summarizing solutions: eval_asym_connectivity_summary(), eval_connectivity_summary(), eval_cost_summary(), eval_feature_representation_summary(), eval_n_summary(), eval_target_coverage_summ

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)
```

```
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()
sim_zones_pu_polygons <- get_sim_zones_pu_polygons()
sim_zones_features <- get_sim_zones_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(verbose = FALSE)

# solve the problem
s1 <- solve(p1)

# print solution
print(s1)

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

# calculate boundary associated with the solution
r1 <- eval_boundary_summary(p1, s1)
print(r1)

# build minimal conservation problem with polygon 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)

# solve the problem
s2 <- solve(p2)

# print solution
print(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 data
p3 <-
  problem(
    sim_zones_pu_polygons, sim_zones_features,
```

```
    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 solution
print(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 connectivity of solution*

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. It is specifically designed for symmetric connectivity data.

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() object.
solution	numeric, matrix, data.frame, terra::rast(), 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 working with multiple zones and the argument to data is a matrix or Matrix object. If the argument to data is an array or data.frame with data for multiple zones (e.g., using the "zone1" and "zone2" column names), 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 the Connectivity_In metric reported by the *Marxan* software (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 to data, weighted by the value of the planning units in the solution.

Value

A 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** terra::rast() **planning units** The argument to solution be a terra::rast() object where different cells 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** `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.

`data` **as a** `matrix`**/**`Matrix` **object** 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` **as a** `data.frame` **object** containing columns that are named `"id1"`, `"id2"`, and `"boundary"`. Here, each row denotes the connectivity between a pair of planning units (per values in the `"id1"` and `"id2"` columns) following the *Marxan* format. If the argument to `x` contains multiple zones, then the `"zone1"` and `"zone2"` columns can optionally be provided to manually specify the connectivity values between planning units when they are allocated to specific zones. If the `"zone1"` and `"zone2"` columns are present, then the argument to `zones` must be `NULL`.

`data` **as an** `array` **object** 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

See summaries for an overview of all functions for summarizing solutions. Also, see add_connectivity_penalties() to penalize solutions with low connectivity.

Other functions for summarizing solutions: eval_asym_connectivity_summary(), eval_boundary_summary(), eval_cost_summary(), eval_feature_representation_summary(), eval_n_summary(), eval_target_coverage_summ

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_polygons <- get_sim_zones_pu_polygons()
sim_zones_features <- get_sim_zones_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(verbose = FALSE)

# solve the problem
s1 <- solve(p1)

# print solution
print(s1)

# plot solution
plot(s1, main = "solution", axes = 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 multi-zone conservation problem with polygon data
p2 <-
  problem(
    sim_zones_pu_polygons, sim_zones_features,
```

```
    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
s2 <- solve(p2)

# print solution
print(s2)

# create new column representing the zone id that each planning unit
# was allocated to in the solution
s2$solution <- category_vector(
  s2[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")]
)
s2$solution <- factor(s2$solution)

# plot solution
plot(s2[, "solution"])

# simulate connectivity matrix
# here, we will add a new column to sim_zones_pu_polygons with
# randomly simulated values and create a connectivity matrix
# based on the average simulated values of adjacent planning units
sim_zones_pu_polygons$con <- runif(nrow(sim_zones_pu_polygons))
cm2 <- connectivity_matrix(sim_zones_pu_polygons, "con")

# calculate connectivity associated with the solution
r2 <- eval_connectivity_summary(
  p2, s2[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")],
  data = cm2
)
print(r2)


## End(Not run)
```

`eval_cost_summary` *Evaluate cost of solution*

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)
```

Arguments

x	problem() object.
solution	numeric, matrix, data.frame, terra::rast(), 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

A 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** terra::rast() **planning units** The argument to solution be a terra::rast() object where different cells 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** 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

See summaries for an overview of all functions for summarizing solutions.

Other functions for summarizing solutions: eval_asym_connectivity_summary(), eval_boundary_summary(), eval_connectivity_summary(), eval_feature_representation_summary(), eval_n_summary(), eval_target_coverage_summary()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()
sim_zones_pu_polygons <- get_sim_zones_pu_polygons()
sim_zones_features <- get_sim_zones_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(verbose = FALSE)

# solve the problem
s1 <- solve(p1)

# print solution
print(s1)

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

# calculate cost of the solution
r1 <- eval_cost_summary(p1, s1)
print(r1)

# build minimal conservation problem with polygon 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)

# solve the problem
s2 <- solve(p2)

# plot solution
plot(s2[, "solution_1"])

# print solution
print(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$solution_1 * sim_pu_polygons$cost, na.rm = TRUE)
print(r2_manual)

# build multi-zone conservation problem with polygon data
p3 <-
  problem(
    sim_zones_pu_polygons, sim_zones_features,
    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 solution
print(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 by solution

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)
```

Arguments

x	`problem()` object.
solution	`numeric`, `matrix`, `data.frame`, `terra::rast()`, 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

A `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 statistics associated with the `"overall"` value in this column are calculated using all planning unit values. For problems with multiple management zones, this means that all calculations are completed by summing together all planning unit values across all zones. For example, if there are two zones, a single planning unit, and a feature has a value of one in the single planning unit for both zones, then `total_amount` will contain a value of two (even though it would not be possible to to achieve a value of two because the planning unit could not simultaneously be allocated to both zones). Additionally, 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** `terra::rast()` **planning units** The argument to `solution` be a `terra::rast()` object where different cells 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 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

See summaries for an overview of all functions for summarizing solutions.

Other functions for summarizing solutions: `eval_asym_connectivity_summary()`, `eval_boundary_summary()`, `eval_connectivity_summary()`, `eval_cost_summary()`, `eval_n_summary()`, `eval_target_coverage_summary()`

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_pu_polygons <- get_sim_zones_pu_polygons()
sim_zones_features <- get_sim_zones_features()

# 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")
  )
)

# solve 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 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)

# build minimal conservation problem with polygon 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)

# solve 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"], main = "solution", axes = FALSE)

# build multi-zone conservation problem with raster data
p4 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  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 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)
```

```
# build multi-zone conservation problem with polygon data
p5 <-
  problem(
    sim_zones_pu_polygons, sim_zones_features,
    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 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).

Usage

```
eval_ferrier_importance(x, solution)
```

Arguments

x	problem() object.
solution	numeric, matrix, data.frame, terra::rast(), 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 that use targets and a single zone. It will throw an error for problems that do not meet these criteria.

Value

A matrix, tibble::tibble(), terra::rast(), or sf::st_sf() 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.

Notes

In previous versions, the documentation for this function had a warning indicating that the mathematical formulation for this function required verification. The mathematical formulation for this function has since been corrected and verified, so now this function is recommended for general use.

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** terra::rast() **planning units** The argument to solution be a terra::rast() object where different cells 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** `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

See importance for an overview of all functions for evaluating the importance of planning units selected in a solution.

Other functions for evaluating solution importance: `eval_rank_importance()`, `eval_rare_richness_importance()`, `eval_replacement_importance()`

Examples

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

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_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)

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

```
# print solution
print(s1)

# plot solution
plot(s1, main = "solution", axes = 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 = names(fs1), axes = FALSE)

# create minimal problem with polygon planning units
p2 <-
  problem(sim_pu_polygons, 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)


## End(Not run)
```

`eval_n_summary` *Evaluate number of planning units selected by solution*

Description

Calculate the number of planning units selected within a solution to a conservation planning problem.

Usage

```
eval_n_summary(x, solution)
```

Arguments

x	problem() object.
solution	numeric, matrix, data.frame, terra::rast(), 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

A 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** terra::rast() **planning units** The argument to solution be a terra::rast() object where different cells 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** 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

See summaries for an overview of all functions for summarizing solutions.

Other functions for summarizing solutions: eval_asym_connectivity_summary(), eval_boundary_summary(), eval_connectivity_summary(), eval_cost_summary(), eval_feature_representation_summary(), eval_target_coverage_summary()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()
sim_zones_pu_polygons <- get_sim_zones_pu_polygons()
sim_zones_features <- get_sim_zones_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(verbose = FALSE)
```

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

# print solution
print(s1)

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

# calculate number of selected planning units within solution
r1 <- eval_n_summary(p1, s1)
print(r1)

# build minimal conservation problem with polygon 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)

# solve the problem
s2 <- solve(p2)

# plot solution
plot(s2[, "solution_1"])

# print solution
print(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_1, na.rm = TRUE)
print(r2_manual)

# build multi-zone conservation problem with polygon data
p3 <-
  problem(
    sim_zones_pu_polygons, sim_zones_features,
    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 solution
```

```
print(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_rank_importance` *Evaluate solution importance using incremental ranks*

Description

Calculate importance scores for the planning units selected in a solution using an incremental rank procedure (based on Jung *et al.* 2021).

Usage

```
eval_rank_importance(
  x,
  solution,
  ...,
  run_checks = TRUE,
  force = FALSE,
  by_zone = TRUE,
  objective = NULL,
  extra_args = NULL,
  n,
  budgets
)
```

Arguments

x	`problem()` object.
solution	`numeric`, `matrix`, `data.frame`, `terra::rast()`, 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.

`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 should be made to solve the problem even if potential issues were detected during the presolve checks. Defaults to `FALSE`.

`by_zone` `logical` value indicating how budgets should be calculated when `x` has multiple zones. If `TRUE`, then the incremental rank procedure will increment budgets for each zone separately. If `FALSE`, then the incremental rank procedure will increment a single budget that is applied to all zones. Note that this parameter is only considered if `n` is specified, and does not affect processing if `budgets` is specified. Defaults to `TRUE`.

`objective` `character` value with the name of the objective function that should be used for the incremental rank procedure. This function must be budget limited (e.g., cannot be `add_min_set_objective()`). For example, `"add_min_shortfall_objective"` can be used to specify the minimum shortfall objective (per `add_min_shortfall_objective()`). Defaults to `NULL` such that the incremental rank procedure will use the objective specified by `x`. If using this default and `x` has the minimum set objective, then the minimum shortfall objective is used (per `add_min_shortfall_objective()`).

`extra_args` `list` with additional arguments for the objective function (excluding the `budget` parameter). For example, this parameter can be used to supply phylogenetic data for the phylogenetic diversity objective function (i.e., when using `objective = "add_max_phylo_div_objective"`). Defaults to `NULL` such that no additional arguments are supplied.

`n` `integer` number of increments for the incremental rank procedure. Note that either `n` or `budgets` (not both) must be specified. If `n` is specified, then `by_zone` is considered during processing for problems with multiple zones.

`budgets` `numeric` vector with budget thresholds for each increment in the incremental rank procedure. Note that either `n` or `budgets` (not both) must be specified.

Details

Importance scores are calculated using an incremental rank procedure. Note that if a problem (per `x`) has complex constraints (i.e., constraints that do not involve locking in or locking out planning units), then the `budgets` parameter must be specified. The incremental rank procedure involves the following steps.

1. A set of budgets are defined. If an argument to the `budgets` parameter is supplied, then the budgets are defined using the `budgets`. Otherwise, if an argument to the `n` parameter is supplied, then the budgets are automatically calculated as a set of values – with equal increments between successive values – that range to a maximum value that is equal to the total cost of `solution`. For example, if considering a problem (per `x`) with a single zone, a solution with a total cost of 400, and `n = 4`: then the budgets will be automatically calculated as 100, 200, 300, and 400. If considering a multiple zone problem and `by_zone = FALSE`, then the budgets will based calculated based on the total cost of the `solution` across all zones. Otherwise if `by_zone = TRUE`, then the budgets are calculated and set based on the total cost of planning

units allocated to each zone (separately) in the `solution`. Note that after running this function, you can see what budgets were defined by accessing attributes from the result (see below for examples).

2. The problem (per `x`) is checked for potential issues. This step is performed to avoid issues during subsequent optimization steps. Note that this step can be skipped using `run_checks = FALSE`. Also, if issues are detected and you wish to proceed anyway, then use`force = TRUE` ignore any detected issues.
3. The problem is modified for subsequent optimization. In particular, the upper bounds for the planning units in the problem are specified based on the `solution`. For problems (per `x`) that have binary decision types, this step is equivalent to locking out any planning units that are not selected in the `solution`. Note that this step is important to ensure that all subsequent optimization processes produce solutions that are nested within the `solution`.
4. The problem is further modified for subsequent optimization. Specifically, its objective is overwritten using the objective defined for the incremental rank procedure (per `objective`) with the budget defined for the first increment. When this step is repeated during subsequent increments, the objective will be overwritten with with the budget defined for the next increment. Additionally, if an argument to the `extra_args` parameter is specified, this argument is used when overwriting the objective.
5. The modified problem is solved to generate a solution. Due to the steps used to modify the problem (i.e., steps 3 and 4), the newly generated solution will contain a subset of the selected planning units in the original `solution`.
6. The status of the planning units in the newly generated solution are recorded for later use (e.g., binary values indicating if planning units were selected or not, or the proportion of each planning unit selected) .
7. The problem is further modified for subsequent optimization. Specifically, the status of the planning units in the newly generated solution are used to set the lower bounds for the planning units in the problem. For problems with binary type decision variables, this step is equivalent to modifying the problem to lock in planning units that were selected by the newly generated solution. Additionally, the newly generated solution is used to specify the starting solution for the subsequent optimization process to reduce processing time (note this is only done when using the *CBC* or *Gurobi* solvers).
8. Steps 4–7 are repeated for each of the remaining budget increments. As increasingly greater budgets are used at higher increments, the modified problem will begin to generate solutions that become increasingly more similar to the original `solution`. Note that the status of the planning units in each of these new solutions are recorded for later use.
9. The incremental optimization rank procedure has now completed. The planning unit solution statuses that were previously recorded in each iteration are used to compute relative importance scores. These relative importance scores range between 0 and 1, with higher scores indicating that a given planning unit was selected in earlier increments and is more cost-effective for meeting the objective (per `objective`). In particular, for a given planning unit, the importance score is calculated based on the arithmetic mean of the status values. For example, if we performed an incremental rank procedure with five increments and binary decision variables, then a planning unit might have been selected in the second increment. In this example, the planning unit would have the following solution statuses across the five increments: (1st increment) 0, (2nd increment) 1, (3rd increment) 1, (4th increment) 1, and (5th increment) 1. The mean of these values is 0.8, and so the planning unit would have an importance score of 0.8. A score of 0.8 is relatively high, and suggests that this planning unit is highly cost-effective.

10. The importance scores are output in the same format as the planning units in the problem (per x) (see the Solution Format section for details).

Value

A numeric, matrix, data.frame, terra::rast(), or sf::sf() object containing importance scores for the planning units in the solution. Specifically, the returned object is in the same format as the planning unit data in the argument to x. The object also has the following attributes that provide information on the incremental rank procedure.

budgets numeric vector or matrix containing the budgets used for each increment in the incremental rank procedure. If the problem (per x) has a single zone, then the budgets are a numeric vector, wherein values correspond to the budgets for each increment. Otherwise, if the problem (per x) has multiple zones, then the budgets are a matrix and their format depends on the by_zone parameter. If by_zone = FALSE, then the budgets are a matrix with a column for each zone and a row for each budget increment. Alternatively, if by_zone = TRUE, then the matrix has a single column and a row for each budget increment.

objective numeric mathematical objective values for each solution generated by the incremental rank procedure.

runtime numeric total amount of time elapsed (reported in seconds) during the optimization process for each solution generated by the incremental rank procedure.

status character status of the optimization process for each solution generated by the incremental rank procedure. See solve() for details on interpreting these values.

gap numeric values describing the optimality gap for each solution generated by the incremental rank procedure. See solve() for details on interpreting these 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** terra::rast() **planning units** The argument to solution be a terra::rast() object where different cells 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** `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

Jung M, Arnell A, de Lamo X, García-Rangel S, Lewis M, Mark J, Merow C, Miles L, Ondo I, Pironon S, Ravilious C, Rivers M, Schepaschenko D, Tallowin O, van Soesbergen A, Govaerts R, Boyle BL, Enquist BJ, Feng X, Gallagher R, Maitner B, Meiri S, Mulligan M, Ofer G, Roll U, Hanson JO, Jetz W, Di Marco M, McGowan J, Rinnan DS, Sachs JD, Lesiv M, Adams VM, Andrew SC, Burger JR, Hannah L, Marquet PA, McCarthy JK, Morueta-Holme N, Newman EA, Park DS, Roehrdanz PR, Svenning J-C, Violle C, Wieringa JJ, Wynne G, Fritz S, Strassburg BBN, Obersteiner M, Kapos V, Burgess N, Schmidt- Traub G, Visconti P (2021) Areas of global importance for conserving terrestrial biodiversity, carbon and water. *Nature Ecology and Evolution*, 5: 1499–1509.

See Also

Other functions for evaluating solution importance: `eval_ferrier_importance()`, `eval_rare_richness_importance()`, `eval_replacement_importance()`

Examples

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

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_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)

# solve problem
s1 <- solve(p1)

# print solution
print(s1)

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

# calculate importance scores using 10 budget increments
# N.B. since the objective for the incremental rank procedure is not
# explicitly defined and the problem has a minimum set objective, the
# the minimum shortfall objective is used by default
rs1 <- eval_rank_importance(p1, s1, n = 10)

# print importance scores
print(rs1)

# plot importance scores
plot(rs1, main = "rank importance (10, min shortfall obj", axes = FALSE)

# display optimization information from the attributes
## status
print(attr(rs1, "status"))
## optimality gap
print(attr(rs1, "gap"))
## run time
print(attr(rs1, "runtime"))
## objective value
print(attr(rs1, "objective"))

# plot relationship between objective values and budget increment
plot(
  y = attr(rs1, "objective"),
  x = seq_along(attr(rs1, "objective")),
  ylab = "objective value", xlab = "budget increment",
  main = "Relationship between objective values and budget increment"
)

# calculate importance scores using the maximum utility objective and
# based on 10 different budgets
rs2 <- eval_rank_importance(
  p1, s1, n = 10, objective = "add_max_utility_objective"
)

# print importance scores
print(rs2)

# plot importance scores
plot(rs2, main = "rank importance (10, max utility obj)", axes = FALSE)
```

```
# calculate importance scores based on 5 manually specified budgets

# calculate 5 ranks using equal intervals
# N.B. we use length.out = 6 because we want 5 budgets > 0
budgets <- seq(0, eval_cost_summary(p1, s1)$cost[[1]], length.out = 6)[-1]

# calculate importance using manually specified budgets
# N.B. since the objective is not explicitly defined and the problem has a
# minimum set objective, the minimum shortfall objective is used by default
rs3 <- eval_rank_importance(p1, s1, budgets = budgets)

# print importance scores
print(rs3)

# plot importance scores
plot(rs3, main = "rank importance (manual)", axes = FALSE)

# build multi-zone conservation problem with raster data
p4 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  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)
names(s4) <- paste0("zone ", seq_len(terra::nlyr(sim_zones_pu_raster)))

# 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, axes = FALSE)

# calculate importance scores
rs4 <- eval_rank_importance(p4, s4, n = 5)
names(rs4) <- paste0("zone ", seq_len(terra::nlyr(sim_zones_pu_raster)))

# plot importance
# each panel corresponds to a different zone, and data show the
# importance of each planning unit in a given zone
plot(rs4, axes = FALSE)

## 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).

Usage

```
eval_rare_richness_importance(x, solution, rescale = TRUE)
```

Arguments

`x`	`problem()` object.
`solution`	`numeric`, `matrix`, `data.frame`, `terra::rast()`, 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.
`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:

$$RWR_k = \sum_{j}^{J} \frac{\frac{r_{ik}}{m_j}}{\sum_i^I r_{ij}}$$

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.

Value

A `numeric`, `matrix`, `data.frame`, `terra::rast()`, 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** `terra::rast()` **planning units** The argument to `solution` be a `terra::rast()` object where different cells 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** `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

See importance for an overview of all functions for evaluating the importance of planning units selected in a solution.

Other functions for evaluating solution importance: `eval_ferrier_importance()`, `eval_rank_importance()`, `eval_replacement_importance()`

Examples

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

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_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)

# 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)

# create minimal problem with polygon planning units
p2 <-
  problem(sim_pu_polygons, 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,
  rescale = TRUE,
  run_checks = TRUE,
  force = FALSE,
  threads = 1L
)
```

Arguments

`x`	`problem()` object.
`solution`	`numeric`, `matrix`, `data.frame`, `terra::rast()`, 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.
`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 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 be 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 recommend 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`, `terra::rast()`, 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** `terra::rast()` **planning units** The argument to `solution` be a `terra::rast()` object where different cells 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** 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

See importance for an overview of all functions for evaluating the importance of planning units selected in a solution.

Other functions for evaluating solution importance: eval_ferrier_importance(), eval_rank_importance(), eval_rare_richness_importance()

Examples

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

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_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)

# solve problem
s1 <- solve(p1)

# print solution
print(s1)

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

# 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)

# create minimal problem with polygon planning units
p3 <-
  problem(sim_pu_polygons, 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_zones_pu_raster, sim_zones_features) %>%
  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)
names(s4) <- paste0("zone ", seq_len(terra::nlyr(s4)))

# 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, axes = FALSE)

# calculate importance scores
rc4 <- eval_replacement_importance(p4, s4)
names(rc4) <- paste0("zone ", seq_len(terra::nlyr(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, axes = FALSE)

## End(Not run)
```

---

eval_target_coverage_summary

Evaluate target coverage by solution

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 = number_of_zones(x) > 1,
  include_sense = number_of_zones(x) > 1
)
```

Arguments

x	problem() object.
solution	numeric, matrix, data.frame, terra::rast(), 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

A 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 threshold amount associated with each target (i.e., `"absolute_target"` 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.

Notes

In prior versions (< 8.0.6.7), this function calculated the relative shortfall for a target by dividing the total shortfall for the target (i.e., `"absolute_shortfall"` column) by the total amount associated with each target (i.e., `"total_amount"` column). This was subsequently changed to ensure consistency with the minimum shortfall objective (`add_min_shortfall_objective()`).

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** `terra::rast()` **planning units** The argument to `solution` be a `terra::rast()` object where different cells 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** `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

See summaries for an overview of all functions for summarizing solutions.

Other functions for summarizing solutions: `eval_asym_connectivity_summary()`, `eval_boundary_summary()`, `eval_connectivity_summary()`, `eval_cost_summary()`, `eval_feature_representation_summary()`, `eval_n_summary()`

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()
```

```
sim_zones_pu_polygons <- get_sim_zones_pu_polygons()
sim_zones_features <- get_sim_zones_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(verbose = FALSE)

# solve the problem
s1 <- solve(p1)

# print solution
print(s1)

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

# 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 data
p3 <-
  problem(
    sim_zones_pu_polygons, sim_zones_features,
    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 solution
print(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 `terra::rast()` object.

Usage

```
fast_extract(x, y, ...)

## S4 method for signature 'Raster,Spatial'
fast_extract(x, y, fun = "mean", ...)

## S4 method for signature 'Raster,sfc'
```

```
fast_extract(x, y, fun = "mean", ...)

## S4 method for signature 'SpatRaster,sfc'
fast_extract(x, y, fun = "mean", ...)

## S4 method for signature 'Raster,sf'
fast_extract(x, y, fun = "mean", ...)

## S4 method for signature 'SpatRaster,sf'
fast_extract(x, y, fun = "mean", ...)
```

Arguments

x	terra::rast() object.
y	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

The performance of this function for large terra::rast() objects can be improved by increasing the GDAL cache size. The default cache size is 25 MB. For example, the following code can be used to set the cache size to 4 GB.

```
terra::gdalCache(size = 4000)
```

This function is simply a wrapper that uses exactextractr::exact_extract() for polygon geometries, and terra::extract() for other geometry types.

Value

A 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

terra::extract(), exactextractr::exact_extract().

Examples

```
# load data
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()

# extract data
result <- fast_extract(sim_features, sim_pu_polygons)
```

```
# 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()` 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

A `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

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_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
s4 <- try(solve(p4))

# 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)

# 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, ...)

## S3 method for class 'ConservationProblem'
feature_names(x, ...)

## S3 method for class 'ZonesRaster'
feature_names(x, ...)

## S3 method for class 'ZonesSpatRaster'
feature_names(x, ...)

## S3 method for class 'ZonesCharacter'
feature_names(x, ...)
```

Arguments

x	`problem()` or `Zones()` object.
...	not used.

Value

A `character` vector of feature names.

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_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))

## End(Not run)
```

`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 functions are available for calculating importance scores for a solution to a conservation planning `problem()`.

`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_rank_importance()` Calculate importance scores using ranks (based on Jung *et al.* 2021). These scores measure importance using an incremental optimization approach. They can (i) account for the cost of different planning units, (ii) account for multiple management zones, and (iii) apply to solutions generated with any objective function.

`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 that use targets and a single zone (e.g., 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.

Broadly speaking, we recommend using replacement cost scores where possible. This is because they can be applied to any type of conservation planning problem – regardless of the objective function or number of zones considered in the problem – and measure planning unit importance based on degradation of the prioritization. Although the replacement cost scores can be calculated for small and moderate sized problems (e.g., less than 30,000 planning units), they may not be feasible for particularly large problems (e.g., more than 100,000 planning units). In such cases, we recommend calculating importance scores using the rank method. This is because it can be calculated relatively quickly for large-sized problems and can explicitly account for costs and representation targets (depending on the objective function used). If using the the rank method with a solution generated using the minimum set objective (i.e., `add_min_set_objective()`), we recommend using the minimum shortfall objective. The Ferrier method can also be useful to highly identify irreplaceable planning units. We only recommend using the rarity weighted richness metric when neither of the other two methods can be used.

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.

Jung M, Arnell A, de Lamo X, García-Rangel S, Lewis M, Mark J, Merow C, Miles L, Ondo I, Pironon S, Ravilious C, Rivers M, Schepaschenko D, Tallowin O, van Soesbergen A, Govaerts R, Boyle BL, Enquist BJ, Feng X, Gallagher R, Maitner B, Meiri S, Mulligan M, Ofer G, Roll U, Hanson JO, Jetz W, Di Marco M, McGowan J, Rinnan DS, Sachs JD, Lesiv M, Adams VM, Andrew SC, Burger JR, Hannah L, Marquet PA, McCarthy JK, Morueta-Holme N, Newman EA, Park DS, Roehrdanz PR, Svenning J-C, Violle C, Wieringa JJ, Wynne G, Fritz S, Strassburg BBN, Obersteiner M, Kapos V, Burgess N, Schmidt- Traub G, Visconti P (2021) Areas of global importance for conserving terrestrial biodiversity, carbon and water. *Nature Ecology and Evolution*, 5: 1499–1509.

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

Other overviews: `constraints`, `decisions`, `objectives`, `penalties`, `portfolios`, `solvers`, `summaries`, `targets`

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_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)

# 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)

# create multi-band raster with different importance scores
ir <- c(ir1, ir2, ir3)
names(ir) <- c(
  "replacement cost", "Ferrier score", "rarity weighted richness"
)

# plot importance scores
plot(ir, axes = FALSE)

## 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,ANY'
intersecting_units(x, y)

## S4 method for signature 'ANY,Raster'
intersecting_units(x, y)

## S4 method for signature 'Spatial,ANY'
```

```
intersecting_units(x, y)

## S4 method for signature 'ANY,Spatial'
intersecting_units(x, y)

## S4 method for signature 'SpatRaster,SpatRaster'
intersecting_units(x, y)

## S4 method for signature 'sf,sf'
intersecting_units(x, y)

## S4 method for signature 'SpatRaster,sf'
intersecting_units(x, y)

## S4 method for signature 'sf,SpatRaster'
intersecting_units(x, y)
```

Arguments

x	sf::st_sf() or terra::rast() object.
y	sf::st_sf() or terra::rast() object.

Details

The performance of this function for large terra::rast() objects can be improved by increasing the GDAL cache size. The default cache size is 25 MB. For example, the following code can be used to set the cache size to 4 GB.

```
terra::gdalCache(size = 4000)
```

Value

An integer vector of indices of the units in x that intersect with y.

See Also

See fast_extract() for extracting data from spatial datasets.

Examples

```
## Not run:
# create data
r <- terra::rast(matrix(1:9, byrow = TRUE, ncol = 3))
r_with_holes <- r
r_with_holes[c(1, 5, 9)] <- NA
ply <- sf::st_as_sf(terra::as.polygons(r))
ply_with_holes <- sf::st_as_sf(terra::as.polygons(r_with_holes))

# intersect raster with raster
par(mfrow = c(1, 2))
```

```
plot(r, main = "x = SpatRaster", axes = FALSE)
plot(r_with_holes, main = "y = SpatRaster", axes = FALSE)
print(intersecting_units(r, r_with_holes))

# intersect raster with sf
par(mfrow = c(1, 2))
plot(r, main = "x = SpatRaster", axes = FALSE)
plot(ply_with_holes, main = "y = sf", key.pos = NULL, reset = FALSE)
print(intersecting_units(r, ply_with_holes))

# intersect sf with raster
par(mfrow = c(1, 2))
plot(ply, main = "x = sf", key.pos = NULL, reset = FALSE)
plot(r_with_holes, main = "y = SpatRaster")
print(intersecting_units(ply, r_with_holes))

# intersect sf with sf
par(mfrow = c(1, 2))
plot(ply, main = "x = sf", key.pos = NULL, reset = FALSE)
plot(ply_with_holes, main = "y = sf", key.pos = NULL, reset = FALSE)
print(intersecting_units(ply, ply_with_holes))

## End(Not run)
```

`knit_print` *Print an object for knitr package.*

Description

This function is used to ensure that `problem()` and `new_optimization_problem()` objects are displayed correctly in **rmarkdown** reports.

Usage

```
knit_print.ConservationProblem(x, ...)

knit_print.OptimizationProblem(x, ...)
```

Arguments

`x`	Object.
`...`	Not used.

Details

This function should not be called directly. It is intended to be used by the **knitr** package when displaying objects.

Value

A `character` vector for knitting.

`linear_interpolation` *Linear interpolation*

Description

Linearly interpolate values between two thresholds.

Usage

```
linear_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

A `numeric` vector.

Examples

```
## Not run:
# 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
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)
```

`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

A `numeric` vector.

Examples

```
## Not run:
# 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
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)

# 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))
)

## End(Not run)
```

`marxan_boundary_data_to_matrix`

Convert Marxan *boundary data to matrix format*

Description

Convert a `data.frame` object containing *Marxan* boundary data to matrix format. This function is designed specifically for boundary data (not connectivity data). It ensures that the output matrix correctly specifies symmetric spatial relationships between planning units.

Usage

```
marxan_boundary_data_to_matrix(x, data)
```

Arguments

`x`	`problem()` 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"`.

Value

A `Matrix::dgCMatrix` sparse matrix object.

Notes

In earlier versions, the function could convert boundary data that pertain to multiple zones. This is no longer possible, following updates to streamline the package.

Examples

```
# create example planning unit layer
pu_data <-
  matrix(1:3, nrow = 1) %>%
  terra::rast() %>%
  setNames("id") %>%
  terra::as.polygons() %>%
  sf::st_as_sf()

# plot planning units
plot(pu_data)

# manually create Marxan boundary data for these planning units following
# the Marxan boundary data format specification
bldf <- data.frame(
  id1 = c(1, 2, 3, 1, 2),
  id2 =c(1, 2, 3, 2, 3),
  boundary = c(3, 2, 3, 1, 1)
)

# print data
print(bldf)

# convert to boundary matrix format for use in prioritizr
m1 <- marxan_boundary_data_to_matrix(NULL, bldf)

# print converted matrix
## we can see that the values in bldf and m1 are different,
## this is because Marxan and prioritizr use different formats
## for storing boundary information
print(m1)

# automatically create boundary data for use in prioritizr,
# by using the boundary_matrix() function
m2 <- boundary_matrix(pu_data)

# print matrix
## we can see that the values in m1 and m2 are exactly the same,
## this is because marxan_boundary_data_to_matrix() automatically
## converts Marxan data to the same format as boundary_matrix()
print(m2)
```

`marxan_connectivity_data_to_matrix`

Convert Marxan *connectivity data to matrix format*

Description

Convert a data.frame object containing *Marxan* connectivity data to matrix format. This function is designed specifically for connectivity data (not boundary data). It ensures that the output matrix correctly specifies symmetric or asymmetric connectivity relationships between planning units.

Usage

```
marxan_connectivity_data_to_matrix(x, data, symmetric = TRUE)
```

Arguments

x	problem() 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".
symmetric	logical does the connectivity data describe symmetric relationships between planning units? If the data contain asymmetric connectivity data, this parameter should be set to FALSE. Defaults to TRUE.

Value

A Matrix::dgCMatrix sparse matrix object.

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# create marxan connectivity data with four planning units and one zone,
# and symmetric connectivity values
bldf1 <- expand.grid(id1 = seq_len(4), id2 = seq_len(4))
bldf1$boundary <- 1
bldf1$boundary[bldf1$id1 == bldf1$id2] <- 0.5

# print data
print(bldf1)

# convert to matrix
m1 <- marxan_connectivity_data_to_matrix(NULL, bldf1)

# print matrix
print(m1)

# visualize matrix
Matrix::image(m1)

# create marxan connectivity data with four planning units and one zone,
# and asymmetric connectivity values
bldf2 <- expand.grid(id1 = seq_len(4), id2 = seq_len(4))
```

```
bldf2$boundary <- runif(nrow(bldf2))
bldf2$boundary[bldf1$id1 == bldf1$id2] <- 0.5

# print data
print(bldf2)

# convert to matrix
m2 <- marxan_connectivity_data_to_matrix(NULL, bldf2, symmetric = FALSE)

# print matrix
print(m2)

# visualize matrix
Matrix::image(m2)

# create marxan connectivity with three planning units and two zones,
# and asymmetric connectivity values
bldf3 <- expand.grid(
  id1 = seq_len(3), id2 = seq_len(3),
  zone1 = c("z1", "z2"),
  zone2 = c("z1", "z2")
)
bldf3$boundary <- runif(nrow(bldf3))
bldf3$boundary[bldf3$id1 == bldf3$id2] <- 0

# print data
print(bldf3)

# convert to array
m3 <- marxan_connectivity_data_to_matrix(NULL, bldf3, symmetric = FALSE)

# print array
print(m3)

## End(Not run)
```

`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). Please note that the **vroom** package is required to import *Marxan* data files.

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, symmetric = TRUE, ...)

## 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.

symmetric logical does the boundary data (i.e., argument to bound) describe symmetric relationships between planning units? If the boundary data contain asymmetric connectivity data, this parameter should be set to FALSE. Defaults to TRUE.

Details

This function is provided as a convenient interface for solving *Marxan* problems using the **prioritizr** package. Note that this function does not support all of the functionality provided by the *Marxan* software. In particular, only the following parameters supported: "INPUTDIR", "SPECNAME", "PUNAME", "PUVSPRNAME", "BOUNDNAME", "BLM", and "ASYMMETRICCONNECTIVITY". Additionally, for the specie columns are considered.

Value

A problem() object.

Notes

In previous versions, this function could not accommodate asymmetric connectivity data. It has now been updated to handle asymmetric connectivity 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.

Serra N, Kockel A, Game ET, Grantham H, Possingham HP, and McGowan J (2020) Marxan User Manual: For Marxan version 2.43 and above. The Nature Conservancy (TNC), Arlington, Virginia, United States and Pacific Marine Analysis and Research Association (PacMARA), Victoria, British Columbia, Canada.

See Also

For more information on the correct format for for *Marxan* input data, see the official *Marxan* website, Ball *et al.* (2009), Serra *et al.* (2020).

Examples

```
# create Marxan problem using Marxan input file
# (note this example requires the vroom package to be installed)
## Not run:
input_file <- system.file("extdata/marxan/input.dat", package = "prioritizr")
p1 <-
  marxan_problem(input_file) %>%
  add_default_solver(verbose = FALSE)

# solve problem
s1 <- solve(p1)

# print solution
head(s1)

# create Marxan problem using data.frames that have been loaded into R
# (note this example also requires the vroom package to be installed)
## load in planning unit data
pu_path <- system.file("extdata/marxan/input/pu.dat", package = "prioritizr")
pu_dat <- vroom::vroom(pu_path)
head(pu_dat)

## load in feature data
spec_path <- system.file(
  "extdata/marxan/input/spec.dat", package = "prioritizr"
)
spec_dat <- vroom::vroom(spec_path)
head(spec_dat)

## load in planning unit vs feature data
puvspr_path <- system.file(
  "extdata/marxan/input/puvspr.dat", package = "prioritizr"
)
puvspr_dat <- vroom::vroom(puvspr_path)
head(puvspr_dat)

## load in the boundary data
bound_path <- system.file(
  "extdata/marxan/input/bound.dat", package = "prioritizr"
)
bound_dat <- vroom::vroom(bound_path)
head(bound_dat)

# create problem without the boundary data
p2 <-
  marxan_problem(pu_dat, spec_dat, puvspr_dat) %>%
  add_default_solver(verbose = FALSE)

# solve problem
s2 <- solve(p2)

# print solution
```

```
head(s2)

# 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, blm = 5) %>%
  add_default_solver(verbose = FALSE)

# solve problem
s3 <- solve(p3)

# print solution
head(s3)

## End(Not run)
```

`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

A `Waiver` object.

Examples

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

# print object
print(w)
```

number_of_features *Number of features*

Description

Extract the number of features in an object.

Usage

```
number_of_features(x, ...)

## S3 method for class 'ConservationProblem'
number_of_features(x, ...)

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

## S3 method for class 'ZonesSpatRaster'
number_of_features(x, ...)

## S3 method for class 'ZonesRaster'
number_of_features(x, ...)

## S3 method for class 'ZonesCharacter'
number_of_features(x, ...)
```

Arguments

x	A problem(), optimization_problem(), or zones() object.
...	not used.

Value

An integer number of features.

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_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))

## End(Not run)
```

`number_of_planning_units` *Number of planning units*

Description

Extract the number of planning units in an object.

Usage

```
number_of_planning_units(x, ...)

## S3 method for class 'ConservationProblem'
number_of_planning_units(x, ...)

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

Arguments

x	`problem()` or `optimization_problem()` object.
...	not used.

Details

The planning units for an object corresponds to the number of entries (e.g., rows, cells) for the planning unit data that do not have missing (`NA`) values for every zone. For example, a single-layer raster dataset might have 90 cells and only two of these cells contain non-missing (`NA`) values. As such, this dataset would have two planning units.

Value

An `integer` number of planning units.

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_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))

## End(Not run)
```

`number_of_total_units` *Number of total units*

Description

Extract the number of total units in an object.

Usage

```
number_of_total_units(x, ...)

## S3 method for class 'ConservationProblem'
number_of_total_units(x, ...)
```

Arguments

`x`	`problem()` object.
`...`	not used.

Details

The total units for an object corresponds to the total number of entries (e.g., rows, cells) for the planning unit data. For example, a single-layer raster dataset might have 90 cells and only two of these cells contain non-missing (`NA`) values. As such, this dataset would have 90 total units and two planning units.

Value

An `integer` number of total units.

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()
```

```
# 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_zones_pu_raster, sim_zones_features) %>%
  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))

## End(Not run)
```

`number_of_zones` *Number of zones*

Description

Extract the number of zones in an object.

Usage

```
number_of_zones(x, ...)

## S3 method for class 'ConservationProblem'
number_of_zones(x, ...)

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

## S3 method for class 'ZonesRaster'
number_of_zones(x, ...)
```

```
## S3 method for class 'ZonesSpatRaster'
number_of_zones(x, ...)

## S3 method for class 'ZonesCharacter'
number_of_zones(x, ...)
```

Arguments

x	problem(), optimization_problem(), or zones() object.
...	not used.

Value

An integer number of zones.

Examples

```
## Not run:
# load data
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# print number of zones in a Zones object
print(number_of_zones(sim_zones_features))
# create problem with multiple zones
p <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  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))

## End(Not run)
```

Objective-class *Objective class*

Description

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

Super class

prioritizr::ConservationModifier -> Objective

Public fields

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

`has_weights` `logical` value indicating if the objective supports feature weights.

Methods

Public methods:

- `Objective$default_weights()`
- `Objective$apply()`
- `Objective$clone()`

Method `default_weights()`**:** Specify default value for the feature weights.

Usage:

```
Objective$default_weights()
```

Returns: A `numeric` value.

Method `apply()`**:** Update an optimization problem formulation.

Usage:

```
Objective$apply(x)
```

Arguments:

`x` `optimization_problem()` object.

Returns: Invisible `TRUE`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other classes: `ConservationModifier-class`, `ConservationProblem-class`, `Constraint-class`, `Decision-class`, `OptimizationProblem-class`, `Penalty-class`, `Portfolio-class`, `Solver-class`, `Target-class`, `TargetMethod-class`, `Weight-class`

`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. Please note that all conservation planning problems require an objective function, and attempting to solve a problem without an objective will result in an error.

Details

The following functions can be used to add an objective to a conservation planning `problem()`. For the vast majority of conservation planning exercises, the minimum set and minimum shortfall objectives are most appropriate. Note that if multiple of these functions are added to a `problem()`, then only the last function added will be used.

`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_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_penalties_objective()` Minimize the penalties associated with a problem as much as possible subject to a budget. This is mainly used when performing hierarchical multi-objective optimization.

`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_features_objective()` Fulfill as many targets as possible while ensuring that the cost of the solution does not exceed a budget.

`add_max_cover_objective()` Represent at least one instance of as many features as possible within a given budget. Note that this objective is not compatible with targets.

`add_max_utility_objective()` Maximize the weighted sum of the features represented by the solution subject to a budget. Note that this objective is not compatible with targets.

See Also

Other overviews: `constraints`, `decisions`, `importance`, `penalties`, `portfolios`, `solvers`, `summaries`, `targets`

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
sim_phylogeny <- get_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)

# solve problems
s <- c(
  solve(p1), solve(p2), solve(p3), solve(p4), solve(p5), solve(p6),
  solve(p7), solve(p8)
)
names(s) <- c(
  "min set", "max coverage", "max features", "min shortfall",
  "min largest shortfall", "max phylogenetic diversity",
  "max phylogenetic endemism", "max utility"
)
# plot solutions
plot(s, axes = FALSE)
```

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

`OptimizationProblem-class`
Optimization problem class

Description

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

Public fields

`ptr` A `Rcpp::Xptr` external pointer. Create a new optimization problem object.

Methods

Public methods:

- `OptimizationProblem$new()`
- `OptimizationProblem$print()`
- `OptimizationProblem$show()`
- `OptimizationProblem$ncol()`
- `OptimizationProblem$nrow()`
- `OptimizationProblem$ncell()`
- `OptimizationProblem$modelsense()`
- `OptimizationProblem$vtype()`
- `OptimizationProblem$obj()`
- `OptimizationProblem$A()`
- `OptimizationProblem$rhs()`
- `OptimizationProblem$sense()`
- `OptimizationProblem$lb()`
- `OptimizationProblem$ub()`
- `OptimizationProblem$number_of_features()`
- `OptimizationProblem$number_of_planning_units()`
- `OptimizationProblem$number_of_zones()`
- `OptimizationProblem$col_ids()`
- `OptimizationProblem$row_ids()`
- `OptimizationProblem$compressed_formulation()`
- `OptimizationProblem$shuffle_columns()`
- `OptimizationProblem$copy()`

- `OptimizationProblem$set_obj()`
- `OptimizationProblem$set_lb()`
- `OptimizationProblem$set_ub()`
- `OptimizationProblem$remove_last_linear_constraint()`
- `OptimizationProblem$append_linear_constraints()`
- `OptimizationProblem$clone()`

Method `new()`**:**

Usage:

```
OptimizationProblem$new(ptr)
```

Arguments:

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

Returns: A new `OptimizationProblem` object.

Method `print()`**:** Print concise information about the object.

Usage:

```
OptimizationProblem$print()
```

Returns: Invisible `TRUE`.

Method `show()`**:** Print concise information about the object.

Usage:

```
OptimizationProblem$show()
```

Returns: Invisible `TRUE`.

Method `ncol()`**:** Obtain the number of columns in the problem formulation.

Usage:

```
OptimizationProblem$ncol()
```

Returns: A `numeric` value.

Method `nrow()`**:** Obtain the number of rows in the problem formulation.

Usage:

```
OptimizationProblem$nrow()
```

Returns: A `numeric` value.

Method `ncell()`**:** Obtain the number of cells in the problem formulation.

Usage:

```
OptimizationProblem$ncell()
```

Returns: A `numeric` value.

Method `modelsense()`**:** Obtain the model sense.

Usage:

```
OptimizationProblem$modelsense()
```

Returns: A `character` value.

Method `vtype()`**:** Obtain the decision variable types.

Usage:

```
OptimizationProblem$vtype()
```

Returns: A `character` vector.

Method `obj()`**:** Obtain the objective function.

Usage:

```
OptimizationProblem$obj()
```

Returns: A `numeric` vector.

Method `A()`**:** Obtain the constraint matrix.

Usage:

```
OptimizationProblem$A()
```

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

Method `rhs()`**:** Obtain the right-hand-side constraint values.

Usage:

```
OptimizationProblem$rhs()
```

Returns: A `numeric` vector.

Method `sense()`**:** Obtain the constraint senses.

Usage:

```
OptimizationProblem$sense()
```

Returns: A `character` vector.

Method `lb()`**:** Obtain the lower bounds for the decision variables.

Usage:

```
OptimizationProblem$lb()
```

Returns: A `numeric` vector.

Method `ub()`**:** Obtain the upper bounds for the decision variables.

Usage:

```
OptimizationProblem$ub()
```

Returns: A `numeric` vector.

Method `number_of_features()`**:** Obtain the number of features.

Usage:

```
OptimizationProblem$number_of_features()
```

Returns: A `numeric` value.

Method `number_of_planning_units()`**:** Obtain the number of planning units.

Usage:

```
OptimizationProblem$number_of_planning_units()
```

Returns: A `numeric` value.

Method `number_of_zones()`**:** Obtain the number of zones.

Usage:

```
OptimizationProblem$number_of_zones()
```

Returns: A `numeric` value.

Method `col_ids()`**:** Obtain the identifiers for the columns.

Usage:

```
OptimizationProblem$col_ids()
```

Returns: A `character` value.

Method `row_ids()`**:** Obtain the identifiers for the rows.

Usage:

```
OptimizationProblem$row_ids()
```

Returns: A `character` value.

Method `compressed_formulation()`**:** Is the problem formulation compressed?

Usage:

```
OptimizationProblem$compressed_formulation()
```

Returns: A `logical` value.

Method `shuffle_columns()`**:** Shuffle the order of the columns in the optimization problem.

Usage:

```
OptimizationProblem$shuffle_columns(order)
```

Arguments:

`order` `integer` vector with new order.

Returns: An `integer` vector with indices to un-shuffle the problem.

Method `copy()`**:** Create a copy of the optimization problem.

Usage:

```
OptimizationProblem$copy()
```

Returns: A new `OptimizationProblem` object .

Method `set_obj()`**:** Set objective coefficients for the decision variables in the optimization problem.

Usage:

```
OptimizationProblem$set_obj(obj)
```

Arguments:

`obj` `numeric` vector.

Returns: An invisible `TRUE` indicating success.

Method `set_lb()`**:** Set lower bounds for the decision variables in the optimization problem.

Usage:

```
OptimizationProblem$set_lb(lb)
```

Arguments:

`lb` `numeric` vector.

Returns: An invisible `TRUE` indicating success.

Method `set_ub()`**:** Set upper bounds for the decision variables in the optimization problem.

Usage:

```
OptimizationProblem$set_ub(ub)
```

Arguments:

`ub` `numeric` vector.

Returns: An invisible `TRUE` indicating success.

Method `remove_last_linear_constraint()`**:** Remove last linear constraint added to a problem.

Usage:

```
OptimizationProblem$remove_last_linear_constraint()
```

Returns: An invisible `TRUE` indicating success.

Method `append_linear_constraints()`**:** Append linear constraints to the optimization problem.

Usage:

```
OptimizationProblem$append_linear_constraints(rhs, sense, A, row_ids)
```

Arguments:

`rhs` `numeric` vector with right-hand-side values.

`sense` `character` vector with constraint sense values (i.e., `"<="`, `">="`, or `"="`).

`A` `Matrix::sparseMatrix()` with constraint coefficients.

`row_ids` `character` vector with identifier for constraints.

Returns: An invisible `TRUE` indicating success.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other classes: `ConservationModifier-class`, `ConservationProblem-class`, `Constraint-class`, `Decision-class`, `Objective-class`, `Penalty-class`, `Portfolio-class`, `Solver-class`, `Target-class`, `TargetMethod-class`, `Weight-class`

`OptimizationProblem-methods` *Optimization problem methods*

Description

These functions are used to query and update a `optimization_problem()`.

Usage

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

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

set_obj(x, obj)

## S4 method for signature 'OptimizationProblem,ANY'
set_obj(x, obj)

set_lb(x, lb)

## S4 method for signature 'OptimizationProblem,ANY'
set_lb(x, lb)

set_ub(x, ub)

## S4 method for signature 'OptimizationProblem,ANY'
set_ub(x, ub)

append_linear_constraints(x, rhs, sense, A, row_ids)

## S4 method for signature 'OptimizationProblem,ANY,ANY,ANY,ANY'
append_linear_constraints(x, rhs, sense, A, row_ids)

remove_last_linear_constraint(x)
```

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

Arguments

x	optimization_problem() object.
obj	numeric vector containing a new linear coefficient for each decision variable in the problem.
lb	numeric vector containing a new lower bound for each decision variable in the problem.
ub	numeric vector containing a new upper bound for each decision variable in the problem.
rhs	numeric vector with the right-hand-side values for new constraints.
sense	character vector with senses for new constraints (i.e., ">=", "<=", or "=" values).
A	Matrix::sparseMatrix() matrix with coefficients for new constraints.
row_ids	character vector with identifiers for new constraints.

Details

The following functions are used to query data.

nrow(x) integer number of rows (constraints).

ncol(x) integer number of columns (decision variables).

ncell(x) integer number of cells.

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

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

obj(x) numeric vector specifying the objective function.

A(x) Matrix::dgCMatrix matrix object defining the problem matrix.

rhs(x) numeric vector with right-hand-side linear constraints

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

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

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

number_of_planning_units(x) integer number of planning units in the problem.

number_of_features(x) integer number of features the problem.

The following functions are used to update data. Note that these functions return an invisible TRUE indicating success.

`set_obj(x, obj)` override the objective in the problem. Here, `obj` is a `numeric` vector containing a new linear coefficient for each decision variable in the problem.

`set_lb(x, lb)` override the variable lower bounds in the problem. Here, `lb` is a `numeric` vector containing a new lower bound.for each decision variable in the problem.

`set_ub(x, ub)` override the variable upper bounds in the problem. Here, `ub` is a `numeric` vector containing a new upper bound.for each decision variable in the problem.

`remove_last_linear_constraint()` remove the last linear constraint added to a problem.

`append_linear_constraints(x, A, sense, rhs, row_ids)` add an additional linear constraints to a problem. Here, `A` is a `Matrix::sparseMatrix()` matrix, `sense` is a `character` vector with constraint senses (i.e., `">="`, `"<="`, or `"="` values), `rhs` is a `numeric` vector with the right-hand-side values, and `row_ids` is a `character` vector with identifiers.

Value

A `Matrix::dgCMatrix`, `numeric` vector, `numeric` vector, or scalar `integer` depending on the method used.

`optimization_problem` *Optimization problem*

Description

Create a new optimization problem.

Usage

```
optimization_problem(x = NULL)
```

Arguments

`x`	A `NULL` or `list` object. See Details for more information. Defaults to `NULL`.

Details

The argument to `x` can be a `NULL` or a `list`. If `x` is a `NULL`, then an empty optimization problem is created. Alternately, if a `x` is a `list` then a fully formulated optimization problem is created. Specifically, the `list` should contain 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.

Value

An OptimizationProblem object.

See Also

OptimizationProblem-methods.

Examples

```
# create new empty object
x1 <- optimization_problem()

# print new empty object
print(x1)

# create list with optimization problem
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 fully formulated object based on lists
x2 <- optimization_problem(l)

# print fully formulated object
print(x2)
```

penalties *Add a penalty*

Description

A penalty can be applied to a conservation planning problem to penalize solutions according to a specific metric. They directly trade-off with the primary objective of a problem (e.g., the primary objective when using `add_min_set_objective()` is to minimize solution cost). If you want to generate a prioritization that only focuses on minimizing a particular penalty, then the minimum penalties objective should be used (i.e., `add_min_penalties_objective()`).

Details

Both penalties and constraints can be used to modify a problem and identify solutions that exhibit specific characteristics. Constraints work by adding rules to ensure that solutions exhibit (or do not exhibit) a particular characteristic. On the other hand, penalties work by specifying trade-offs against the primary problem objective and are mediated by a penalty factor. Note that although multiple penalty functions can be added to a `problem()`, this will likely increase solver run time.

The following functions can be used to add a penalty 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_neighbor_penalties()` Add penalties to a conservation problem to favor solutions that have a large number of planning units sited next to each other. This constraints may be especially useful for reducing spatial fragmentation in large-scale planning problems or when using open source solvers.

`add_asym_connectivity_penalties()` Add penalties to a conservation problem to account for asymmetric connectivity.

`add_connectivity_penalties()` Add penalties to a conservation problem to account for symmetric connectivity.

`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

For information on calibrating the penalties, see the Calibrating Trade-offs vignette. Also, see `calibrate_cohon_penalty()` for assistance with selecting an appropriate `penalty` value.

Other overviews: `constraints`, `decisions`, `importance`, `objectives`, `portfolios`, `solvers`, `summaries`, `targets`

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()
```

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

# create problem with boundary penalties
p2 <- p1 %>% add_boundary_penalties(5, 1)

# create problem with neighbor penalties
p3 <- p1 %>% add_neighbor_penalties(4)

# create connectivity matrix based on spatial proximity
scm <- terra::as.data.frame(sim_pu_raster, xy = TRUE, na.rm = FALSE)
scm <- 1 / (as.matrix(dist(as.matrix(scm))) + 1)

# remove weak and moderate connections between planning units to reduce
# run time
scm[scm < 0.85] <- 0

# create problem with connectivity penalties
p4 <- p1 %>% add_connectivity_penalties(25, data = scm)

# create asymmetric connectivity data by randomly simulating values
acm <- matrix(runif(ncell(sim_pu_raster) ^ 2), ncol = ncell(sim_pu_raster))
acm[acm < 0.85] <- 0

# create problem with asymmetric connectivity penalties
p5 <- p1 %>% add_asym_connectivity_penalties(1, data = acm)

# 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)

# create problem with linear penalties, with a penalty scaling factor of 100
p6 <- p1 %>% add_linear_penalties(100, data = sim_penalty_raster)

# solve problems
s <- c(
  solve(p1), solve(p2), solve(p3), solve(p4), solve(p5), solve(p6)
)
names(s) <- c(
  "basic solution", "boundary penalties", "neighbor penalties",
  "connectivity penalties", "asymmetric penalties", "linear penalties"
)
```

```
# plot solutions
plot(s, axes = FALSE)

## End(Not run)
```

`Penalty-class` *Penalty class*

Description

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

Super class

`prioritizr::ConservationModifier` -> `Penalty`

Methods

Public methods:

- `Penalty$apply()`
- `Penalty$clone()`

Method `apply()`**:** Update an optimization problem formulation.

Usage:

```
Penalty$apply(x)
```

Arguments:

`x` `optimization_problem()` object.

Returns: Invisible `TRUE`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other classes: `ConservationModifier-class`, `ConservationProblem-class`, `Constraint-class`, `Decision-class`, `Objective-class`, `OptimizationProblem-class`, `Portfolio-class`, `Solver-class`, `Target-class`, `TargetMethod-class`, `Weight-class`

Portfolio-class *Portfolio class*

Description

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

Super class

`prioritizr::ConservationModifier` -> `Portfolio`

Methods

Public methods:

- `Portfolio$run()`
- `Portfolio$clone()`

Method `run()`**:** Run the portfolio to generate solutions.

Usage:

```
Portfolio$run(x, solver)
```

Arguments:

`x` `optimization_problem()` object.

`solver` `Solver` object.

Returns: `list` of solutions.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other classes: `ConservationModifier-class`, `ConservationProblem-class`, `Constraint-class`, `Decision-class`, `Objective-class`, `OptimizationProblem-class`, `Penalty-class`, `Solver-class`, `Target-class`, `TargetMethod-class`, `Weight-class`

`portfolios` *Add 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 functions can be used to add a portfolio to a conservation planning `problem()`. Note that if multiple of these functions are added to a `problem()`, then only the last function added will be used.

`add_default_portfolio()` Generate a portfolio containing a single solution. This portfolio method is added to `problem()` objects by default.

`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. Although this method may be useful for generating multiple solutions that can be used to calculate selection frequencies (similar to *Marxan*), note that this function tends to produce solutions that are very similar to each other and so you will (likely) need to generate thousands for solutions when calculating selection frequencies. 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

Other overviews: `constraints`, `decisions`, `importance`, `objectives`, `penalties`, `solvers`, `summaries`, `targets`

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_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 default portfolio
p1 <- p %>% add_default_portfolio()

# create problem with cuts portfolio with 4 solutions
p2 <- p %>% add_cuts_portfolio(4)

# create problem with shuffle portfolio with 4 solutions
p3 <- p %>% add_shuffle_portfolio(4)

# create problem with extra portfolio
p4 <- p %>% add_extra_portfolio()

# create problem with top portfolio with 4 solutions
p5 <- p %>% add_top_portfolio(4)

# create problem with gap portfolio with 4 solutions within 50% of optimality
p6 <- p %>% add_gap_portfolio(4, 0.5)

# solve problems to obtain solution portfolios
s <- list(solve(p1), solve(p2), solve(p3), solve(p4), solve(p5), solve(p6))

# plot solution from default portfolio
plot(terra::rast(s[[1]]), axes = FALSE)

# plot solutions from cuts portfolio
plot(terra::rast(s[[2]]), axes = FALSE)

# plot solutions from shuffle portfolio
plot(terra::rast(s[[3]]), axes = FALSE)

# plot solutions from extra portfolio
plot(terra::rast(s[[4]]), axes = FALSE)

# plot solutions from top portfolio
plot(terra::rast(s[[5]]), axes = FALSE)

# plot solutions from gap portfolio
plot(terra::rast(s[[6]]), axes = FALSE)
```

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

`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, warn = TRUE)

## S3 method for class 'ConservationProblem'
presolve_check(x, warn = TRUE)

## S3 method for class 'OptimizationProblem'
presolve_check(x, warn = TRUE)
```

Arguments

`x`	`problem()` or `optimization_problem()` object.
`warn`	`logical` should a warning be thrown if the presolve checks fail? Defaults to `TRUE`.

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

A `logical` value indicating if all checks passed successfully.

See Also

problem(), solve(), http://www.gurobi.cn/download/GuNum.pdf.

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_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))

# 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)

## End(Not run)
```

`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 (Hanson *et al.* 2025). 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 (e.g., Rodrigues *et al.* 2000; Billionnet 2013). 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 (Schuster *et al.* 2020). 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 Hanson *et al.* (2025) and 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 Brief introduction to systematic conservation planning and demonstration of the main package features.

package_overview Comprehensive introduction to systematic conservation planning and detailed overview of the package features.

calibrating_trade-offs_tutorial Examples of balancing different criteria to identify candidate prioritizations.

connectivity_tutorial Examples of incorporating and evaluating connectivity in prioritizations using a range of approaches.

management_zones_tutorial Tutorial on using multiple management actions or zones to create detailed prioritizations.

gurobi_installation Instructions for installing and setting up the *Gurobi* optimization software for use with the package.

solver_benchmark Reports run times for solving conservation planning problems of varying size and complexity using different solvers.

publication_record List of publications that have cited the package.

Citation

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

Hanson JO, Schuster R, Strimas-Mackey M, Morrell N, Edwards BPM, Arcese P, Bennett JR, and Possingham HP (2025) Systematic conservation prioritization with the prioritizr R package. *Conservation Biology*, 39: e14376.

Author(s)

Authors:

- Jeffrey O Hanson <jeffrey.hanson@uqconnect.edu.au> (ORCID)
- Richard Schuster <richard.schuster@glel.carleton.ca> (ORCID, maintainer)
- Nina Morrell <nina.morrell@ubc.ca>
- Matthew Strimas-Mackey <mstrimas@gmail.com> (ORCID)
- Brandon P M Edwards <brandonedwards3@cmail.carleton.ca> (ORCID)
- Matthew E Watts <m.watts@uq.edu.au>
- Peter Arcese <peter.arcese@ubc.ca> (ORCID)
- Joseph Bennett <joseph.bennett@carleton.ca> (ORCID)
- Hugh P Possingham <hugh.possingham@tnc.org> (ORCID)

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.

Hanson JO, Schuster R, Strimas-Mackey M, Morrell N, Edwards BPM, Arcese P, Bennett JR, and Possingham HP (2025) Systematic conservation prioritization with the prioritizr R package. *Conservation Biology*, 39: e14376.

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.

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

Useful links:

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

`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(...)

distribute_load(...)

new_optimization_problem(...)

predefined_optimization_problem(...)

add_loglinear_targets(...)
```

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.

`add_loglinear_targets()` replaced by the `spec_interp_absolute_targets()` function.

`set_number_of_threads()` no longer needed due to improved data extraction methods.

`get_number_of_threads()` no longer needed due to improved data extraction methods.

`is.parallel()` no longer needed due to improved data extraction methods.

`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.

`distribute_load()` has been removed because it is no longer used. See `parallel::splitIndices()` for equivalent functionality.

`new_optimization_problem()` replaced by `optimization_problem()`.

`predefined_optimization_problem()` replaced by `optimization_problem()`.

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 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 'SpatRaster,SpatRaster'
problem(x, features, run_checks, ...)

## S4 method for signature 'SpatRaster,ZonesSpatRaster'
problem(x, features, run_checks, ...)

## S4 method for signature 'data.frame,character'
problem(x, features, cost_column, feature_units, ...)

## S4 method for signature 'data.frame,ZonesCharacter'
problem(x, features, cost_column, feature_units, ...)

## S4 method for signature 'data.frame,data.frame'
problem(x, features, rij, cost_column, zones, feature_units, ...)

## S4 method for signature 'numeric,data.frame'
problem(x, features, rij_matrix, feature_units, ...)

## S4 method for signature 'matrix,data.frame'
problem(x, features, rij_matrix, feature_units, ...)

## S4 method for signature 'sf,SpatRaster'
problem(x, features, cost_column, run_checks, ...)

## S4 method for signature 'sf,ZonesSpatRaster'
problem(x, features, cost_column, run_checks, ...)

## S4 method for signature 'sf,character'
problem(x, features, cost_column, feature_units, ...)

## S4 method for signature 'sf,ZonesCharacter'
problem(x, features, cost_column, feature_units, ...)
```

```
## 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, feature_units, ...)

## S4 method for signature 'Spatial,ZonesCharacter'
problem(x, features, cost_column, feature_units, ...)

## 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, ...)
```

Arguments

x — terra::rast(), sf::st_sf(), data.frame, matrix, or numeric vector 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_constraints() 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 **has** terra::rast() **or** sf::st_sf() **planning units** The argument to features can be a terra::rast() 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 **has** sf::st_sf() **or** data.frame **planning units** The argument to features can be a character vector with column names (from x) 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 **has** `data.frame`**,** `matrix`**, or** `numeric` **vector planning units** The argument to `features` can be a `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 **has** `terra::rast()` **or** `sf::st_sf()` **planning units** The argument to `features` can be a `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 **has** `sf::st_sf()` **or** `data.frame` **planning units** The argument to `features` can be a `ZonesCharacter` object with column names (from `x`) 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, 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 `sf::st_sf()` 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.

`feature_units` `character` vector containing the unit of measurement for the data associated with each feature. Although the data associated with a feature can any unit, the `feature_units` must refer to area-based units (e.g., `"km^2"`, `"ha"`, `"acre"`). If a feature does not ture_unitsmust not be specified iffeatureshas [terra::rast()] data. Defaults toNULL, such value is used for each feature.

`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 features 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 Matrix::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 Matrix::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 systematic conservation planning exercise leverages data to help inform conservation decision making. To help ensure that the data – and resulting prioritizations – are relevant to the over-arching goals of the exercise, you should decide on the management action (or set of actions) that need be considered in the exercise. For example, these actions could include establishing protected areas, selecting land for conservation easements, restoring habitat, planting trees for carbon sequestration, eradicating invasive species, or some combination of the previous actions. If the exercise involves multiple different actions, they can be incorporated by using multiple zones (see the Management Zones vignette for details). After deciding on the management action(s), you can compile the following data.

First, you will need to create a set of planning units (i.e., discrete spatial areas) to inform decision making. Planning units are often created by subdividing a study region into a set of square or hexagonal cells. They can also be created using administrative boundaries (e.g., provinces), land management boundaries (e.g., property boundaries derived from cadastral data), or ecological boundaries (e.g., based on ecosystem classification data). The size (i.e., spatial grain) of the planning units is often determined based on a compromise between the scale needed to inform decision making, the spatial accuracy (resolution) of available datasets, and the computational resources available for generating prioritizations (e.g., RAM and number of CPU cores on your computer).

Second, you will need data to quantify the cost of implementing each management action within each planning unit. Critically, the cost data should reflect the management action(s) considered in the exercise. For example, costs are often specified using data that reflect economic expenditure (e.g., land acquisition cost), socioeconomic conditions (e.g., human population density), opportunity costs of foregone commercial activities (e.g., logging or agriculture), or opportunity costs of foregone recreational activities (e.g., recreational fishing). In some cases – depending on the management action(s) considered – it can make sense to use a constant cost value (e.g., all planning units are assigned a cost value equal to one) or use a cost value based on spatial extent (e.g., each planning unit is assigned a cost value based on its total area). Also, in most cases, you want to avoid negative cost values. This is because a negative value means that a place is *desirable* for im-

plementing a management action, and such places will almost always be selected for prioritization even if they provide no benefit.

Third, you will need data to quantify the benefits of implementing management actions within planning units. To achieve this, you will need to select a set of conservation features that relate to the over-arching goals of the exercise. For example, conservation features often include species (e.g., Clouded Leopard), habitats (e.g., mangroves or cloud forest), or ecosystems. The benefit that each feature derives from a planning unit can take a variety of forms, but is typically occupancy (i.e., presence or absence), area of occurrence within each planning unit (e.g., based on species' geographic range data), or a measure of habitat suitability (e.g., estimated using a statistical model). After compiling these data, you have the minimal data needed to generate a prioritization.

A systematic conservation planning exercise involves prioritizing a set of management actions to be implemented within certain planning units. Critically, this prioritization should ideally optimize the trade-off between benefits and costs. To accomplish this, the **prioritizr** package uses input data to formulate optimization problems (see Optimization section for details). Broadly speaking, the goal of an optimization problem is to minimize (or maximize) an objective function over a set of decision variables, subject to a series of constraints. Here, an objective function specifies the metric for evaluating conservation plans. The decision variables are what we control, and usually there is one binary variable for each planning unit to specify whether that unit is selected or not (but other approaches are available, see decisions). The constraints can be thought of as rules that must be followed. For example, constraints can be used to ensure a prioritization must stay within a certain budget. These constraints can also leverage additional data to help ensure that prioritizations meet the over-arching goals of the exercise. For example, to account for existing conservation efforts, you could obtain data delineating the extent of existing protected areas and use constraints to lock in planning units that are covered by them (see add_locked_in_constraints).

Value

A new `problem()` (`ConservationProblem`) object.

Optimization

The **prioritizr** package uses exact algorithms to solve reserve design problems (see solvers for details). To achieve this, it internally formulates mathematical optimization problems using mixed integer linear programming (MILP). The general form of such problems can be expressed in matrix notation using the following equation.

$$Minimize \mathbf{c^T x} subject to \mathbf{A x} \geq = or \leq \mathbf{b}$$

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. If you wish to see exactly how a conservation planning problem is formulated as mixed integer linear programming problem, you can use the `write_problem()` function to save the optimization problem to a plain-text file on your computer and then view it using a standard text editor (e.g., Notepad).

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()` function). As a consequence, it can take a while to initialize large-scale conservation planning problems that involve millions of planning units.

See Also

See `solve()` for details on solving a problem to generate solutions. Also, see objectives, penalties, targets, constraints, decisions, portfolios, solvers for information on customizing problems. Additionally, see summaries and importance for information on evaluating solutions.

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_pu_polygons <- get_sim_pu_polygons()
sim_pu_points <- get_sim_pu_points()
sim_pu_lines <- get_sim_pu_lines()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_pu_polygons <- get_sim_zones_pu_polygons()
sim_zones_features <- get_sim_zones_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)

# create problem using polygon 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 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 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)

# since geo-processing can be slow for large spatial vector datasets
# (e.g., polygons, lines, points), it can be worthwhile to pre-process the
# planning unit data so that it contains columns indicating the amount of
# each feature inside each planning unit
# (i.e., each column corresponds to a different feature)

# calculate the amount of each species within each planning unit
pre_proc_data <- rij_matrix(sim_pu_polygons, sim_features)

# add extra columns to the polygon planning unit data
# to indicate the amount of each species within each planning unit
pre_proc_data <- as.data.frame(t(as.matrix(pre_proc_data)))
names(pre_proc_data) <- names(sim_features)
sim_pu_polygons <- cbind(sim_pu_polygons, pre_proc_data)

# create problem using the polygon planning unit data
# with the pre-processed columns
p5 <-
  problem(sim_pu_polygons, features = names(pre_proc_data), "cost") %>%
  add_min_set_objective() %>%
  add_relative_targets(0.2) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# in addition to spatially explicit data, pre-processed aspatial data
# can also be used to create a problem
# (e.g., data created using external spreadsheet software)
costs <- sim_pu_polygons$cost
features <- data.frame(
  id = seq_len(terra::nlyr(sim_features)),
  name = names(sim_features)
)
rij_mat <- rij_matrix(sim_pu_polygons, sim_features)
p6 <-
  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)

# plot solutions for problems associated with spatial data
```

```
plot(s1, main = "raster data", axes = FALSE)
plot(s2[, "solution_1"], main = "polygon data")
plot(s3[, "solution_1"], main = "line data")
plot(s4[, "solution_1"], main = "point data")
plot(s5[, "solution_1"], main = "preprocessed data (polygon data)")

# show solutions for problems associated with aspatial data
str(s6)

# 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_zones_features),
  ncol = number_of_zones(sim_zones_features),
  dimnames = list(
    feature_names(sim_zones_features), zone_names(sim_zones_features)
  )
)

# print targets
print(targets)

# create a multi-zone problem with raster data
p7 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  add_min_set_objective() %>%
  add_absolute_targets(targets) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s7 <- solve(p7)

# plot solution
# here, each layer/panel corresponds to a different zone and cell values
# indicate if a given planning unit has been allocated to a given zone
par(mfrow = c(1, 1))
plot(s7, main = c("zone 1", "zone 2", "zone 3"), axes = 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(s7), axes = FALSE)

# create a multi-zone problem with polygon data
p8 <-
```

```
  problem(
    sim_zones_pu_polygons, sim_zones_features,
    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
s8 <- solve(p8)

# create column containing the zone id for which each planning unit was
# allocated to in the solution
s8$solution <- category_vector(sf::st_drop_geometry(
 s8[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")]
))
s8$solution <- factor(s8$solution)

# plot solution
plot(s8[, "solution"], axes = FALSE)

# create a multi-zone problem with polygon planning unit data
# and where columns correspond to feature abundances

# to begin with, we will add columns to the planning unit data
# that indicate the amount of each feature in each zone
sim_zones_pu_polygons$spp1_z1 <- rpois(nrow(sim_zones_pu_polygons), 1)
sim_zones_pu_polygons$spp2_z1 <- rpois(nrow(sim_zones_pu_polygons), 1)
sim_zones_pu_polygons$spp3_z1 <- rpois(nrow(sim_zones_pu_polygons), 1)
sim_zones_pu_polygons$spp1_z2 <- rpois(nrow(sim_zones_pu_polygons), 1)
sim_zones_pu_polygons$spp2_z2 <- rpois(nrow(sim_zones_pu_polygons), 1)
sim_zones_pu_polygons$spp3_z2 <- rpois(nrow(sim_zones_pu_polygons), 1)

# create problem with polygon planning unit data and use column names
# to indicate feature data
# additionally, to make this example slightly more interesting,
# the problem will have proportion-type decisions such that
# a proportion of each planning unit can be allocated to each of the
# two management zones
p9 <-
  problem(
    sim_zones_pu_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
s9 <- solve(p9)

# plot solution
plot(s9[, c("solution_1_z1", "solution_1_z2")], axes = 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 'SpatRaster'
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`	`terra::rast()` 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

A `Matrix::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 `terra::rast()` object, then cells with missing (`NA`) values are set to zero too.

See Also

Proximity matrix data might need rescaling to improve optimization performance, see `rescale_matrix()` to perform these calculations.

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_pu_polygons <- get_sim_pu_polygons()
sim_pu_lines <- get_sim_pu_lines()
sim_pu_points <- get_sim_pu_points()

# create proximity matrix using raster data
## crop raster to 9 cells to provide a small example
r <- terra::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 data
## subset 9 polygons to provide a small example
ply <- sim_pu_polygons[c(1:3, 11:13, 20:22), ]

## make proximity matrix using a distance threshold of 2
cm_ply <- proximity_matrix(ply, distance = 2)

# create proximity matrix using line data
## subset 9 lines to provide a small example
lns <- sim_pu_lines[c(1:3, 11:13, 20:22), ]

## make proximity matrix
cm_lns <- proximity_matrix(lns, distance = 2)

## create proximity matrix using point data
## subset 9 points to provide a small example
pts <- sim_pu_points[c(1:3, 11:13, 20:22), ]

# make proximity matrix
```

```
cm_pts <- proximity_matrix(pts, distance = 2)

## plot raster and proximity matrix
plot(r, main = "raster", axes = FALSE)
Matrix::image(cm_raster, main = "proximity matrix")

## plot polygons and proximity matrix
plot(ply[, 1], main = "polygons", axes = FALSE)
Matrix::image(cm_ply, main = "proximity matrix")

## plot lines and proximity matrix
plot(lns[, 1], main = "lines", axes = FALSE)
Matrix::image(cm_lns, main = "proximity matrix")

## plot points and proximity matrix
plot(pts[, 1], main = "points", axes = FALSE)
Matrix::image(cm_pts, main = "proximity matrix")

## End(Not run)
```

`rescale_matrix` *Rescale a matrix*

Description

Linearly rescale a matrix. Specifically, the values in the matrix are rescaled so that the maximum value in the matrix is equal to a new user-specified maximum value.

Usage

```
rescale_matrix(x, max = 1000)
```

Arguments

x	matrix, array, Matrix::Matrix object.
max	numeric new maximum value in matrix. Defaults to 1000.

Details

This function is particularly useful for rescaling data prior to optimization to avoid numerical issues. For example, boundary length (e.g., generated using `boundary_matrix()`) or connectivity data (e.g., generated using `connectivity_matrix()`) can contain very large values (e.g., values greater than 1,000,000) and such large values can, in turn, degrade the performance of exact algorithm solvers (see Details section in `presolve_check()` for more information on numerical issues). By using this function to rescale boundary length or connectivity data prior to optimization (e.g., before using `add_boundary_penalties()` or `add_connectivity_penalties()`, this can help avoid numerical issues during optimization.

Value

A matrix, array, or Matrix::Matrix object. The returned object is the is the same class as the argument to x.

See Also

See boundary_matrix() and connectivity_matrix() for details on creating boundary length and connectivity data. Also, see presolve_check() for information on numerical issues.

Examples

```
## Not run:
# rescale_matrix() is especially useful for re-scaling boundary length data
# prior to optimization, and so here we provide an example showing how
# this can be accomplished

# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()

# compute boundary data
bd <- boundary_matrix(sim_pu_raster)

# re-scale boundary data
bd <- rescale_matrix(bd)

# create problem with boundary penalties
p <-
  problem(sim_pu_raster, sim_features) %>%
  add_min_set_objective() %>%
  add_boundary_penalties(0.01, data = bd) %>%
  add_relative_targets(0.2) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s <- solve(p)

# plot solution
plot(s)

## 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 'SpatRaster,SpatRaster'
rij_matrix(x, y, memory, idx, ...)

## S4 method for signature 'sf,SpatRaster'
rij_matrix(x, y, fun, memory, idx, ...)

## S4 method for signature 'Raster,Raster'
rij_matrix(x, y, memory, idx, ...)

## S4 method for signature 'sf,Raster'
rij_matrix(x, y, fun, memory, idx, ...)

## S4 method for signature 'Spatial,Raster'
rij_matrix(x, y, fun, memory, idx, ...)
```

Arguments

x	terra::rast() or sf::sf() object representing planning units.
y	terra::rast() object.
...	not used.
memory	logical should calculations be performed using a method that prioritizes reduced memory consumption over speed? This is useful when processing particularly large raster datasets. If TRUE, then calculations are performed by processing each raster layer in y in a sequential manner. If FALSE, then calculations are performed by processing all raster layers in y together. If NA, then the memory requirements will be estimated and, if required, processing will be performed using the method that reduces memory consumption. Defaults to NA.
idx	integer vector containing planning unit indices. Defaults to NULL such that the indices are computed automatically based on x.
fun	character for summarizing values inside each planning unit. This parameter is only used when the argument to x is a sf::sf() object. Defaults to "sum".

Details

Generally, processing `sf::st_sf()` data takes much longer to process than `terra::rast()` data. As such, it is recommended to use `terra::rast()` data for planning units where possible. The performance of this function for large `terra::rast()` datasets can be improved by increasing the GDAL cache size. The default cache size is 25 MB. For example, the following code can be used to set the cache size to 4 GB.

```
terra::gdalCache(size = 4000)
```

Value

A `Matrix::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

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_pu_polygons <- get_sim_pu_polygons()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_features <- get_sim_features()

# 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 planning units
rij_polygons <- rij_matrix(sim_pu_polygons, sim_features)
print(rij_polygons)

# create rij matrix using raster planning units with multiple zones
rij_zones_raster <- rij_matrix(sim_zones_pu_raster, sim_features)
print(rij_zones_raster)

## End(Not run)
```

---

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() 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

An invisible TRUE indicating success.

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()

# 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)
```

---

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 'OptimizationProblem'
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

Generates simulated cost data using Gaussian random fields.

Usage

```
simulate_cost(x, n, intensity, sd, scale)

## S3 method for class 'Raster'
simulate_cost(x, n = 1, intensity = 100, sd = 20, scale = 2.5)

## S3 method for class 'SpatRaster'
simulate_cost(x, n = 1, intensity = 100, sd = 20, scale = 2.5)
```

Arguments

x	terra::rast() object to use as a template.
n	integer number of layers to simulate. Defaults to 1.
intensity	numeric average value of simulated data. Defaults to 100.
sd	numeric standard deviation of simulated data. Defaults to 20.
scale	numeric parameter to control level of spatial auto-correlation in the simulated data. Defaults to 2.5.

Value

A terra::rast() object with integer values greater than zero.

See Also

Other functions for simulating data: simulate_data(), simulate_species()

Examples

```
## Not run:
# create raster
r <- terra::rast(
  ncols = 10, nrows = 10, xmin = 0, xmax = 1, ymin = 0, ymax = 1, vals = 1
)

# simulate data
cost <- simulate_cost(r)

# plot simulated species
plot(cost, main = "simulated cost data", axes = FALSE)

## End(Not run)
```

`simulate_data` *Simulate data*

Description

Simulate spatially auto-correlated data using Gaussian random fields.

Usage

```
simulate_data(x, n, scale, intensity, sd, transform)

## S3 method for class 'Raster'
simulate_data(
  x,
  n = 1,
  scale = 0.5,
  intensity = 0,
  sd = 1,
  transform = identity
)

## S3 method for class 'SpatRaster'
simulate_data(
  x,
  n = 1,
  scale = 0.5,
  intensity = 0,
  sd = 1,
  transform = identity
)
```

Arguments

x	terra::rast() object to use as a template.
n	integer number of layers to simulate. Defaults to 1.
scale	numeric parameter to control level of spatial auto-correlation in the simulated data. Defaults to 0.5.
intensity	numeric average value of simulated data. Defaults to 0.
sd	numeric standard deviation of simulated data. Defaults to 1.
transform	function transform values output from the simulation. Defaults to the identity() function such that values remain the same following transformation.

Value

A terra::rast() object.

See Also

Other functions for simulating data: simulate_cost(), simulate_species()

Examples

```
## Not run:
# create raster
r <- terra::rast(
  ncols = 10, nrows = 10, xmin = 0, xmax = 1, ymin = 0, ymax = 1, vals = 1
)

# simulate data using a Gaussian field
x <- simulate_data(r, n = 1, scale = 0.2)

# plot simulated data
plot(x, main = "simulated data", axes = FALSE)

## End(Not run)
```

---

simulate_species *Simulate species habitat suitability data*

Description

Generates simulated species data using Gaussian random fields.

Usage

```
simulate_species(x, n, scale)

## S3 method for class 'Raster'
simulate_species(x, n = 1, scale = 0.5)

## S3 method for class 'SpatRaster'
simulate_species(x, n = 1, scale = 0.5)
```

Arguments

x	terra::rast() object to use as a template.
n	integer number of layers to simulate. Defaults to 1.
scale	numeric parameter to control level of spatial auto-correlation in the simulated data. Defaults to 0.5.

Value

A terra::rast() object with values between zero and one.

See Also

Other functions for simulating data: simulate_cost(), simulate_data()

Examples

```
## Not run:
# create raster
r <- terra::rast(
  ncols = 10, nrows = 10, xmin = 0, xmax = 1, ymin = 0, ymax = 1, vals = 1
)

# simulate data for 4 species
spp <- simulate_species(r, 4)

# plot simulated species
plot(spp, main = "simulated species distributions", axes = FALSE)

## End(Not run)
```

---

sim_data *Get simulated conservation planning data*

Description

A set of functions are available for importing simulated datasets. These datasets are designed for creating small example spatial prioritizations.

Usage

```
get_sim_pu_polygons()

get_sim_zones_pu_polygons()

get_sim_pu_lines()

get_sim_pu_points()

get_sim_pu_raster()

get_sim_locked_in_raster()

get_sim_locked_out_raster()

get_sim_zones_pu_raster()

get_sim_features()

get_sim_zones_features()

get_sim_phylogeny()

get_sim_complex_pu_raster()

get_sim_complex_locked_in_raster()

get_sim_complex_locked_out_raster()

get_sim_complex_features()

get_sim_complex_historical_features()
```

Format

`get_sim_pu_polygons()` `sf::st_sf()` object.
`get_sim_zones_pu_polygons()` `sf::st_sf()` object.
`get_sim_pu_lines()` `sf::st_sf()` object.
`get_sim_pu_points()` `sf::st_sf()` object.
`get_sim_pu_raster()` `terra::rast()` object.
`get_sim_zones_pu_raster()` `terra::rast()` object.
`get_sim_locked_in_raster()` `terra::rast()` object.
`get_sim_locked_out_raster()` `terra::rast()` object.
`get_sim_features()` `terra::rast()` object.
`get_sim_zones_features()` `ZonesRaster()` object.
`get_sim_phylogeny()` `terra::rast()` object.

Small-scale single zone datasets

The following functions are provided for generating small-scale spatial prioritizations that only contain a single management zone.

`get_sim_pu_raster()` Import planning unit data that are stored in raster format. Here, cell values indicate planning unit cost and missing (`NA`) values indicate that a cell is not a planning unit.

`get_sim_locked_in_raster()` Import planning unit data that are stored in raster format. Here, cell values are binary and indicate if planning units should be locked in to a solution.

`get_sim_locked_out_raster()` Import planning unit data that are stored in raster format. Here, cell values are binary and indicate if planning units should be locked out from a solution.

`get_sim_pu_polygons()` Import planning unit data stored in vector format. Here, planning units are represented using spatial polygons (e.g., each polygon corresponds to a different management areas). The data contains columns indicating the expenditure required for prioritizing each planning unit (`"cost"` column), if the planning units should be selected in the solution (`"locked_in"` column), and if the planning units should never be selected in the solution (`"locked_out"` column).

`get_sim_pu_points()` Import planning unit data stored in vector format. Here, planning units are represented using spatial lines (e.g., each line corresponds to a different section along a river) . The attribute table follows the same conventions as for `sim_pu_polygons`.

`get_sim_pu_lines()` Import planning unit data stored in vector format. Here, planning units are represented using spatial points (e.g., each point corresponds to a different site) . The attribute table follows the same conventions as for `sim_pu_polygons`.

`get_sim_features()` Import feature data stored in raster format. Here, data describe the spatial distribution of five species. Each layer corresponds to a different species, and cell values indicate habitat suitability.

`get_sim_phylogeny()` Import phylogenetic tree for the five species.

Complex single zone datasets

The following functions are provided for generating spatial prioritizations based on a complex dataset that contains a single management zone.

`get_sim_complex_pu_raster()` Import planning unit data that are stored in raster format. Here, cell values indicate planning unit cost and missing (`NA`) values indicate that a cell is not a planning unit.

`get_sim_complex_locked_in_raster()` Import planning unit data that are stored in raster format. Here, cell values are binary and indicate if planning units should be locked in to a solution.

`get_sim_complex_locked_out_raster()` Import planning unit data that are stored in raster format. Here, cell values are binary and indicate if planning units should be locked out from a solution.

`get_sim_complex_features()` Import feature data stored in raster format. Here, data describe the spatial distribution of 100 species. Each layer corresponds to a different species, and cells contain binary values indicating if they currently contain habitat for a given species.

`get_sim_complex_historical_features()` Import potential feature data stored in raster format. Here, data describe the spatial distribution of 100 species. Each layer corresponds to a different species, and cells contain binary values indicating if they historically contained habitat for a given species.

Multiple zone datasets

The following functions are provided for generating spatial prioritizations that contain multiple management zones.

`get_sim_zones_pu_raster()` Import planning unit data for multiple management zones that are stored in raster format. Here, each layer indicates the cost for a different management zone. Cells with `NA` values in a given zone indicate that a planning unit cannot be allocated to that zone in a solution. Additionally, cells with `NA` values in all layers are not a planning unit.

`get_sim_zones_pu_polygons()` Import planning unit data for multiple management zones stored in vector format. Here, planning units are represented using spatial polygons. The data contains columns indicating the expenditure required for prioritizing each planning unit under different management zones (`"cost_1"`, `"cost_2"`, and `"cost_3"` columns), and a series of columns indicating the value that each planning unit that should be assigned in the solution (`"locked_1"`, `"locked_2"`, `"locked_3"` columns). In these locked columns, planning units that should not be locked to a specific value are assigned a missing (`NA`) value.

`get_sim_zones_features()` Import feature data for multiple management zones stored in raster format. Here, data describe the spatial distribution of ten species under three different management zones.

Examples

```
# load data
sim_pu_polygons <- get_sim_pu_polygons()
sim_zones_pu_polygons <- get_sim_zones_pu_polygons()
sim_pu_lines <- get_sim_pu_lines()
sim_pu_points <- get_sim_pu_points()
sim_pu_raster <- get_sim_pu_raster()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_locked_in_raster <- get_sim_locked_in_raster()
sim_locked_out_raster <- get_sim_locked_out_raster()
sim_phylogeny <- get_sim_phylogeny()
sim_features <- get_sim_features()
sim_zones_features <- get_sim_zones_features()

# plot raster data
## Not run:
par(mfrow = c(2, 2))
plot(sim_pu_raster, main = "planning units (raster)", axes = FALSE)
plot(sim_locked_in_raster, main = "locked in units (raster)", axes = FALSE)
plot(sim_locked_out_raster, main = "locked out units (raster)", axes = FALSE)

# plot vector planning unit data
par(mfrow = c(1, 1))
plot(sim_pu_polygons)
plot(sim_pu_lines)
```

```
plot(sim_pu_points)

# plot vector planning unit data for multiple management zones
plot(sim_zones_pu_polygons)

# plot phylogeny data
par(mfrow = c(1, 1))
plot(sim_phylogeny, main = "simulated phylogeny")

# plot feature data
par(mfrow = c(1, 1))
plot(sim_features, axes = FALSE)

# plot cost data for multiple management zones
par(mfrow = c(1, 1))
plot(sim_zones_pu_raster, axes = FALSE)

# plot feature data for multiple management zones
plot_names <- paste0(
  "Species ",
   rep(
     seq_len(number_of_zones(sim_zones_features)),
     number_of_features(sim_zones_features)
   ),
   " (zone ",
   rep(
     seq_len(number_of_features(sim_zones_features)),
     each = number_of_zones(sim_zones_features)
   ),
   ")"
)
plot(
  terra::rast(as.list(sim_zones_features)),
   main = plot_names, axes = FALSE
)

## End(Not run)
```

`solve` *Solve*

Description

Solve a conservation planning problem.

Usage

```
## S3 method for class 'ConservationProblem'
solve(a, b, ..., run_checks = TRUE, force = FALSE)
```

Arguments

a	problem() object.
b	missing.
...	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").

Value

A numeric, matrix, data.frame, sf::st_sf(), or terra::rast() object containing the solution to the problem. Additionally, the returned object has attributes that describe optimization process or solution (see below for examples on accessing these attributes). These attributes provide the following information.

objective numeric mathematical objective value for the solution used to evaluate the prioritization during optimization.

runtime numeric total amount of time elapsed while during the optimization process (reported in seconds). Note that this measure of time does not include any data pre-processing or post-processing steps.

status character status of the optimization process. This status typically describes the reason why the optimization process terminated. For example, it might indicate that the optimization process terminated because an optimal solution was found, or because a pre-specified time limit was reached. These status values are (mostly) obtained directly from the solver software, and so we recommend consulting the solver's documentation for further information on what particular status values mean. Note that some solvers (e.g., Gurobi and HiGHS) will return an "OPTIMAL" status when the solver has found a solution within the pre-specified optimality gap (e.g., it has found a solution within 10% of optimality), even though the solution itself may not be strictly optimal.

`gap` `numeric` optimality of the solution. This gap value provides an upper bound of how far the solution is from optimality. For example, you might specify a 10% optimality gap for the optimization process (e.g., using `add_highs_solver(gap = 0.1)`), and this might produce a solution that is actually 5% from optimality. As such, the solution might have a gap value of 0.05 (corresponding to 5%). Because this value represents an upper bound, it is also possible that the solution in this example – even though it is actually 5% from optimality – might have a gap value of 7% (i.e., 0.07). Note that only some solvers are able to provide this information (i.e., the *Gurobi* and *HiGHS* solvers), and other solvers will have a missing (`NA`) value.

`objbound` `numeric` best known bound of the optimal objective. Note that only some solvers are able to provide this information (i.e., the *Gurobi* solvers), and other solvers will have a missing (`NA`) value.

Output format

This function will output solutions in a similar format to the planning units associated with `a`. Specifically, it will return solutions based on the following types of planning units.

`a` **has** `numeric` **planning units** The solution will be returned as a `numeric` vector. Here, each element in the vector corresponds to a different planning unit. Note that if a portfolio is used to generate multiple solutions, then a `list` of such `numeric` vectors will be returned.

`a` **has** `matrix` **planning units** The solution will be returned as a `matrix` object. Here, rows correspond to different planning units, and columns correspond to different management zones. Note that if a portfolio is used to generate multiple solutions, then a `list` of such `matrix` objects will be returned.

`a` **has** `terra::rast()` **planning units** The solution will be returned as a `terra::rast()` object. If the argument to `x` contains multiple zones, then the object will have a different layer for each management zone. Note that if a portfolio is used to generate multiple solutions, then a `list` of `terra::rast()` objects will be returned.

`a` **has** `sf::sf()`**, or** `data.frame` **planning units** The solution will be returned in the same data format as the planning units. Here, each row corresponds to a different planning unit, and columns contain solutions. If the argument to `a` contains a single zone, then the solution object will contain columns named by solution. Specifically, the column names containing the solution values be will named as `"solution_XXX"` where `"XXX"` corresponds to a solution identifier (e.g., `"solution_1"`). If the argument to `a` contains multiple zones, then the columns containing solutions will be named as `"solution_XXX_YYY"` where `"XXX"` corresponds to the solution identifier and `"YYY"` is the name of the management zone (e.g., `"solution_1_zone1"`).

See Also

See `problem()` to create conservation planning problems, and `presolve_check()` to check problems for potential issues. Also, see the `category_layer()` and `category_vector()` function to reformat solutions that contain multiple zones.

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)
```

```
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_pu_polygons <- get_sim_zones_pu_polygons()
sim_zones_features <- get_sim_zones_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(verbose = FALSE)

# 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"))
print(attr(s1, "gap"))

# calculate feature representation in the solution
r1 <- eval_feature_representation_summary(p1, s1)
print(r1)

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

# build minimal conservation problem with polygon 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)

# solve the problem
s2 <- solve(p2)

# print solution
print(s2)

# calculate feature representation in the solution
r2 <- eval_feature_representation_summary(p2, s2[, "solution_1"])
print(r2)
```

```
# plot solution
plot(s2[, "solution_1"], main = "solution", axes = FALSE)

# build multi-zone conservation problem with raster data
p3 <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  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 solution
print(s3)

# calculate feature representation in the solution
r3 <- eval_feature_representation_summary(p3, s3)
print(r3)

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

# build multi-zone conservation problem with polygon data
p4 <-
  problem(
    sim_zones_pu_polygons, sim_zones_features,
    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
s4 <- solve(p4)

# print solution
print(s4)

# calculate feature representation in the solution
r4 <- eval_feature_representation_summary(
  p4, s4[, c("solution_1_zone_1", "solution_1_zone_2", "solution_1_zone_3")]
)
print(r4)

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

```
s4$solution <- factor(s4$solution)

# plot solution
plot(s4[, "solution"])

## End(Not run)
```

`Solver-class` *Solver class*

Description

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

Super class

`prioritizr::ConservationModifier` -> `Solver`

Methods

Public methods:

- `Solver$run()`
- `Solver$calculate()`
- `Solver$set_variable_ub()`
- `Solver$set_variable_lb()`
- `Solver$set_constraint_rhs()`
- `Solver$set_start_solution()`
- `Solver$remove_start_solution()`
- `Solver$solve()`
- `Solver$clone()`

Method `run()`**:** Run the solver to generate a solution.

Usage:

```
Solver$run()
```

Returns: `list` of solutions.

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

Usage:

```
Solver$calculate(...)
```

Arguments:

`...` Additional arguments.

`x` `optimization_problem()` object.

Returns: Invisible `TRUE`.

Method `set_variable_ub()`**:** Set the upper bound for a decision variable.

Usage:

```
Solver$set_variable_ub(index, value)
```

Arguments:

`index` `integer` value indicating the index of the decision variable.

`value` `numeric` new bound value.

Details: Note that this method should only be run after `$calculate()`. It can be used to overwrite values after ingesting an `optimization_problem()` object. It is designed to be used in portfolios and importance functions.

Returns: Invisible `TRUE`.

Method `set_variable_lb()`**:** Set the lower bound for a decision variable.

Usage:

```
Solver$set_variable_lb(index, value)
```

Arguments:

`index` `integer` value indicating the index of the decision variable.

`value` `numeric` new bound value.

Details: Note that this method should only be run after `$calculate()`. It can be used to overwrite values after ingesting an `optimization_problem()` object. It is designed to be used in portfolios and importance functions.

Returns: Invisible `TRUE`.

Method `set_constraint_rhs()`**:** Set the right-hand-side coefficient bound for a constraint.

Usage:

```
Solver$set_constraint_rhs(index, value)
```

Arguments:

`index` `integer` value indicating the index of the decision variable.

`value` `numeric` new value.

Details: Note that this method should only be run after `$calculate()`. It can be used to overwrite values after ingesting an `optimization_problem()` object. It is designed to be used in portfolios and importance functions.

Returns: Invisible `TRUE`.

Method `set_start_solution()`**:** Set the starting solution.

Usage:

```
Solver$set_start_solution(value, warn = TRUE)
```

Arguments:

`value` `numeric` vector.

`warn` `logical` indicating if a warning should be displayed if the solver does not support starting solutions.

Details: This method is designed used in portfolios and importance functions.

Returns: Invisible `TRUE`.

Method `remove_start_solution()`**:** Remove the starting solution.

Usage:

```
Solver$remove_start_solution()
```

Details: This method is designed used in portfolios and importance functions.

Returns: Invisible `TRUE`.

Method `solve()`**:** Solve an optimization problem.

Usage:

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

Arguments:

`x` `optimization_problem()` object.

`...` Additional arguments passed to the `calculate()` method.

Returns: Invisible `TRUE`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other classes: `ConservationModifier-class`, `ConservationProblem-class`, `Constraint-class`, `Decision-class`, `Objective-class`, `OptimizationProblem-class`, `Penalty-class`, `Portfolio-class`, `Target-class`, `TargetMethod-class`, `Weight-class`

`solvers` *Add solvers*

Description

Specify the software and settings 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 Hanson *et al.* (2025) and Schuster *et al.* (2020) for benchmarks comparing run time and solution quality of these solvers.

Details

The following functions can be used to add a solver to a conservation planning `problem()`. Note that if multiple of these functions are added to a `problem()`, then only the last function added will be used.

`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_highs_solver()` *HiGHS* is an open source optimization software. Although this solver can have comparable performance to the *CBC* solver for particular problems and is generally faster than the *SYMPHONY* based solvers (see below), it sometimes can take much longer than the *CBC* solver for particular problems.

`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* and *HiGHS* solvers 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. It is not recommended to use this solver because it has the slowest performance.

References

Hanson JO, Schuster R, Strimas-Mackey M, Morrell N, Edwards BPM, Arcese P, Bennett JR, and Possingham HP (2025). Systematic conservation prioritization with the prioritizr R package. *Conservation Biology*, 39: e14376.

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

Other overviews: `constraints`, `decisions`, `importance`, `objectives`, `penalties`, `portfolios`, `summaries`, `targets`

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_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 names
n <- c()

# create empty list to store solutions
s <- c()

# if gurobi is installed: create problem with added gurobi solver
if (require("gurobi")) {
  p1 <- p %>% add_gurobi_solver(verbose = FALSE)
  n <- c(n, "gurobi")
  s <- c(s, solve(p1))
}

# if cplexAPI is installed: create problem with added CPLEX solver
if (require("cplexAPI")) {
  p2 <- p %>% add_cplex_solver(verbose = FALSE)
  n <- c(n, "CPLEX")
  s <- c(s, solve(p2))
}

# if rcbc is installed: create problem with added CBC solver
if (require("rcbc")) {
  p3 <- p %>% add_cbc_solver(verbose = FALSE)
  n <- c(n, "CBC")
  s <- c(s, solve(p3))
}

# if highs is installed: create problem with added HiGHs solver
if (require("highs")) {
  p4 <- p %>% add_highs_solver(verbose = FALSE)
  n <- c(n, "HiGHS")
  s <- c(s, solve(p4))
}

# create problem with added rsymphony solver
if (require("Rsymphony")) {
  p5 <- p %>% add_rsymphony_solver(verbose = FALSE)
  n <- c(n, "Rsymphony")
  s <- c(s, solve(p5))
```

```
}

# if lpsymphony is installed: create problem with added lpsymphony solver
if (require("lpsymphony")) {
  p6 <- p %>% add_lpsymphony_solver(verbose = FALSE)
  n <- c(n, "lpsymphony")
  s <- c(s, solve(p6))
}

# plot solutions
names(s) <- n
plot(terra::rast(s), axes = FALSE)

## End(Not run)
```

`spec_absolute_targets` *Specify absolute targets*

Description

Specify targets expressed as the same values as the underlying feature data (ignoring any specified feature units). For example, setting a target of 10 for a feature specifies that a solution should ideally select a set of planning units that contain a total (summed) value of, at least, 10 for the feature. This function is designed to be used with `add_auto_targets()`.

Usage

```
spec_absolute_targets(targets, ...)
```

Arguments

`targets`	`numeric` vector that specifies targets for each of the features. If a single value is specified, then all features are assigned the same target threshold.
`...`	not used.

Value

An object (`TargetMethod`) for specifying targets that can be used with `add_auto_targets()` and `add_group_targets()` to add targets to a `problem()`.

Mathematical formulation

This method involves setting target thresholds based on a pre-specified value. To express this mathematically, we will define the following terminology. Let a the absolute target for a feature (per `targets`). Given this terminology, the target threshold (t) for the feature is calculated as follows.

$$t = a$$

See Also

To add relative targets directly to a problem(), see add_absolute_targets().

Other target setting methods: spec_area_targets(), spec_duran_targets(), spec_interp_absolute_targets(), spec_interp_area_targets(), spec_jung_targets(), spec_max_targets(), spec_min_targets(), spec_polak_targets(), spec_pop_size_targets(), spec_relative_targets(), spec_rl_ecosystem_targets(), spec_rl_species_targets(), spec_rodrigues_targets(), spec_rule_targets(), spec_ward_targets(), spec_watson_targets(), spec_wilson_targets()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()

# create base problem
p0 <-
  problem(sim_pu_raster, sim_features) %>%
  add_min_set_objective() %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# this function sets targets based on the total abundance of the features
# (i.e., sum of planning unit values for the feature) and does not
# consider the spatial area covered by the planning units

# create problem with absolute targets of 5 for each feature
p1 <-
  p0 %>%
  add_auto_targets(method = spec_absolute_targets(targets = 5))

# solve problem
s1 <- solve(p1)

# plot solution
plot(s1, main = "solution based on constant targets", axes = FALSE)

# targets can also be specified for each feature separately.
# to demonstrate this, we will set a target value for each
# feature based on a random number between 1 and 5
target_values <- runif(terra::nlyr(sim_features), 1, 5)

# create problem with targets defined separately for each feature
p2 <-
  p0 %>%
  add_auto_targets(method = spec_absolute_targets(targets = target_values))

# solve problem
```

```
s2 <- solve(p2)

# plot solution
plot(s2, main = "solution based on varying targets", axes = FALSE)

## End(Not run)
```

`spec_area_targets` *Specify targets based on area units*

Description

Specify targets expressed as area-based units. For example, this function can be used to express targets as hectares, acres, or km^2. To ensure feasibility, area-based targets are clamped based on the total abundance of features.

Usage

```
spec_area_targets(targets, area_units)
```

Arguments

`targets`	`numeric` vector denoting the area-based targets for the features. These values must be expressed in the same units as `area_units`. If a single `numeric` value is specified, then all features are assigned targets based on the area-based target.
`area_units`	`character` vector denoting the unit of measurement for each `targets` value (e.g., `"km^2"`, `"ha"`, and `"acres"`). If a single `character` value is specified, then all features are assigned targets assuming that `targets` are in the same units.

Value

An object (`TargetMethod`) for specifying targets that can be used with `add_auto_targets()` and `add_group_targets()` to add targets to a `problem()`.

Mathematical formulation

This method provides an approach for setting target thresholds based on an area-based threshold. To express this mathematically, we will define the following terminology. Let f denote the total spatial extent of a feature (e.g., geographic range size expressed as km^2), and a the specified area-based target (expressed as km^2, per `targets` and `area_units`). Given this terminology, the target threshold (t) for the feature is calculated as follows.

$$t = min(f, a)$$

Data calculations

This function involves calculating targets based on the spatial extent of the features in `x`. Although it can be readily applied to `problem()` objects that have the feature data provided as a `terra::rast()` object, you will need to specify the spatial units for the features when initializing the `problem()` objects if the feature data are provided in a different format. In particular, if the feature data are provided as a `data.frame` or `character` vector, then you will need to specify an argument to `feature_units` when using the `problem()` function. See the Examples section of the documentation for `add_auto_targets()` for a demonstration of specifying the spatial units for features.

See Also

Other target setting methods: `spec_absolute_targets()`, `spec_duran_targets()`, `spec_interp_absolute_targets()`, `spec_interp_area_targets()`, `spec_jung_targets()`, `spec_max_targets()`, `spec_min_targets()`, `spec_polak_targets()`, `spec_pop_size_targets()`, `spec_relative_targets()`, `spec_rl_ecosystem_targets()`, `spec_rl_species_targets()`, `spec_rodrigues_targets()`, `spec_rule_targets()`, `spec_ward_targets()`, `spec_watson_targets()`, `spec_wilson_targets()`

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_complex_pu_raster <- get_sim_complex_pu_raster()
sim_complex_features <- get_sim_complex_features()

# create base problem
p0 <-
  problem(sim_complex_pu_raster, sim_complex_features) %>%
  add_min_set_objective() %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# create problem with targets of 50 km^2 for each feature
p1 <-
  p0 %>%
  add_auto_targets(
    method = spec_area_targets(targets = 50, area_units = "km^2")
  )

# solve problem
s1 <- solve(p1)

# plot solution
plot(s1, main = "solution based on constant targets", axes = FALSE)

# targets can also be specified for each feature separately.
# to demonstrate this, we will set a target value for each
# feature based on a random number between 5000 and 30000 hectares
target_values <- runif(terra::nlyr(sim_complex_features), 5000, 30000)
```

```
# create problem with targets defined separately for each feature
p2 <-
  p0 %>%
  add_auto_targets(
    method = spec_area_targets(targets = target_values, area_units = "ha")
  )

# solve problem
s2 <- solve(p2)

# plot solution
plot(s2, main = "solution based on varying targets", axes = FALSE)

## End(Not run)
```

`spec_duran_targets` *Specify targets following Durán* et al. *(2020)*

Description

Specify targets based on the methodology outlined by Durán *et al.* (2020). Briefly, this method involves using historical distribution data to infer the minimum amount of habitat required for a species to have a particular probability of persisting indefinitely. Note that this function is designed to be used with `add_auto_targets()` and `add_group_targets()`.

Usage

```
spec_duran_targets(probability_target, historical_area, area_units)
```

Arguments

`probability_target`
: `numeric` vector denoting the minimum probability of persistence for each feature. For example, a value of 0.1 corresponds to a 10% chance of persistence, and a value of 1 corresponds to a 100% chance of persistence. Values must range between 0 and 1. If a single `numeric` value is specified, then all features are assigned targets assuming the same probability threshold.

`historical_area`
: `numeric` vector denoting the total area encompassed by the historical distribution for each feature (e.g., in km^2). If a single `numeric` value is specified, then all features are assigned targets assuming the same historical distribution size. See Details section for information on obtaining these data.

`area_units`
: `character` vector denoting the unit of measurement for the argument to `historical_area`. For example, to specify that `historical_area` contains historical distribution sizes expressed as km^2, then `area_units = "km^2"` should be used. If a single `character` value is specified, then all features are assigned targets assuming the same area units.

Details

This target setting method is derived from a framework for estimating the impacts of anthropogenic activities at national and global scales (Durán *et al.* 2020). It involves setting targets based on an estimate of the minimum amount of habitat required for a species to have a particular probability of persistence. Although the gold standard approach for estimating such an amount of habitat would involve population viability analysis (e.g., Taylor *et al.* 2017), population viability analyses require a considerable amount of species-specific data that are often not available for conservation planning exercises (reviewed by Akçakaya and Sjögren-Gulve 2000). As such, this method provides a less data intensive alternative for setting targets based on desired probabilities of persistence. Please note that this function is provided as convenient method to set targets for problems with a single management zone, and cannot be used for those with multiple management zones.

This target setting method involves calculating species representation targets based on aspirational goals for species recovery and persistence. For example, let's consider setting `probability_target = 0.95`. If a species has a large amount of its historical distribution remaining, then this probability target threshold may result in setting a species representation target that is equivalent to 90% of the species' current distribution. Assuming that the assumptions that underpin this target setting method are correct (see below), such a probability target threshold would seek to prevent future habitat loss from causing this species to have a chance of persistence below 95%. Additionally, if a threatened species has a relatively small amount of its historical distribution remaining and not enough habitat for this species remains to achieve a 95% chance of persistence, then this probability target threshold may result in a setting species representation target that is equivalent to 100% of the species' current distribution. In this case – given the assumptions of this target setting method (see below) – such a probability target threshold would seek to enable future species recovery efforts to secure a 95% chance of persistence.

To use this method effectively, probability target thresholds (i.e., `probability_targets`) will need to be set carefully. One option for setting such thresholds could be based on probabilities estimated for threat statuses associated with the Red List of Threatened Species by the International Union for the Conservation of Nature (IUCN). For example, Davis et al. (2018) estimated that species recognized as Least Concern on the Red List of Threatened Species by the International Union for the Conservation of Nature (IUCN) would have a 0.9983 probability of persistence (i.e., 99.83% chance of persistence). If less a less ambitious goal is more practical, then probabilities of persistence for other threat statuses could be more appropriate (e.g., such as the probability estimated for the Vulnerable threat status). Similar to Davis et al. (2018), Gumbs et al. (2023) also estimated probabilities of extinction for threat statuses recognized the IUCN Red List of Threatened Species. For reference, we provide the probabilities of persistence estimated by Davis et al. (2018) and Gumbs et al. (2023) below.

Davis et al. (2018) estimated the following probabilities of persistence.

- Least Concern (LC) has `probability_targets = 0.9983`.
- Near Threatened (NT) has `probability_targets = 0.9859`.
- Vulnerable (VU) has `probability_targets = 0.9`.
- Endangered (EN) has `probability_targets = 0.3277`.
- Critically Endangered (CR) has `probability_targets = 0.001`.

Gumbs et al. (2023) estimated the following probabilities of persistence.

- Least Concern (LC) has `probability_targets = 0.939375`.

- Near Threatened (NT) has `probability_targets` = `0.87875`.
- Vulnerable (VU) has `probability_targets` = `0.7575`.
- Endangered (EN) has `probability_targets` = `0.515`.
- Critically Endangered (CR) has `probability_targets` = `0.03`.

This target setting method relies heavily on assumptions. In particular, it is based on the assumption that – for a given species – there is an idealized distribution size (e.g., geographic range size) that would allow for the species to have a 100% chance of persisting indefinitely, and decreases in this distribution size would be associated with reductions in the species' probability of persistence (Payne and Finnegan 2007; Purvis *et al.* 2000). Building on this assumption, it further assumes that (i) the historical distribution of a species can reliably approximate its idealized distribution size with a 100% chance of persisting indefinitely, and (ii) proportionate decreases in the species' distribution size (relative to its idealized distribution size) are associated with increasingly greater reductions in probability of persistence (i.e., following a power-law function with an exponent of 0.25) (Balmford *et al.* 2018; Brooks *et al.* 1999; Thomas *et al.* 2004). Based on these assumptions, this method involves estimating the minimum amount of habitat required to ensure that a species has, at least, a particular probability of persistence, and then setting the species' representation target accordingly.

The validity of this target setting method depends on how well its assumptions are justified. As such, great care should be taken to ensure that the historical distribution of a species used to approximate its idealized distribution does indeed have a (near) 100% probability of persistence (Durán et al. 2020). Although the package does not provide historical distribution data, such data can be derived from species distribution modeling techniques. For example, one approach for characterizing the historical distribution of a species is to fit an environmental niche model based on present-day environmental data and then use historical environmental data to predict the species' historical distribution (reviewed by Nogués-Bravo 2009). Another approach involves using the area of habitat framework (reviewed by Brooks *et al.* 2019) with information on a species' habitat preferences, current and former (i.e., now extinct) geographic ranges, and historical land cover data (Eyres *et al.* 2025).

Value

An object (`TargetMethod`) for specifying targets that can be used with `add_auto_targets()` and `add_group_targets()` to add targets to a `problem()`.

Mathematical formulation

This method involves setting target thresholds based on the area required to ensure that each feature meets a pre-specified probability of persistence. To express this mathematically, we will define the following terminology. Let f denote the total abundance of a feature (e.g., current geographic range size expressed as km^2), h the historical total abundance of the feature (e.g., historical range size expressed as km^2, per `historical_area` and `area_units`), and p the desired threshold probability of persistence for the feature. Given this terminology, the target threshold (t) for the feature is calculated as follows.

$$t = min(f, h \times p^{\frac{1}{0.25}})$$

Data calculations

This function involves calculating targets based on the spatial extent of the features in `x`. Although it can be readily applied to `problem()` objects that have the feature data provided as a `terra::rast()`

object, you will need to specify the spatial units for the features when initializing the `problem()` objects if the feature data are provided in a different format. In particular, if the feature data are provided as a `data.frame` or `character` vector, then you will need to specify an argument to `feature_units` when using the `problem()` function. See the Examples section of the documentation for `add_auto_targets()` for a demonstration of specifying the spatial units for features.

References

Akçakaya HR, Sjögren-Gulve P (2000) Population viability analyses in conservation planning: an overview. *Ecological Bulletins*, 48:9–21. Balmford B, Green RE, Onial M, Phalan B, Balmford A (2018) How imperfect can land sparing be before land sharing is more favourable for wild species? *Journal of Applied Ecology*, 56:73–84.

Brooks TM, Pimm SL, Akçakaya HR, Buchanan GM, Butchart SHM, Foden W, Hilton-Taylor C, Hoffmann M, Jenkins CN, Joppa L, Li BV, Menon V, Ocampo-Peñuela N, Rondinini C (2019) Measuring terrestrial area of habitat (AOH) and its utility for the IUCN Red List. *Trends in Ecology and Evolution*, 34:977–986

Brooks TM, Pimm SL, Oyugi JO (1999) Time lag between deforestation and bird extinction in tropical forest fragments. *Conservation Biology*, 13:1140–1150.

Davis M, Faurby S, Svenning J-C (2018) Mammal diversity will take millions of years to recover from the current biodiversity crisis. *Proceedings of the National Academy of Sciences*, 115:11262–11267.

Durán AP, Green JMH, West CD, Visconti P, Burgess ND, Virah-Sawmy M, Balmford A (2020) A practical approach to measuring the biodiversity impacts of land conversion. *Methods in Ecology and Evolution*, 11:910–921.

Eyres A, Ball TS, Dales M, Swinfield T, Arnell A, Baisero D, Durán AP, Green JMH, Green RE, Madhavapeddy A, Balmford A (2025) LIFE: A metric for mapping the impact of land-cover change on global extinctions. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 380:20230327.

Gumbs R, Gray CL, Böhm M, Burfield IJ, Couchman OR, Faith DP, Forest F, Hoffmann M, Isaac NJB, Jetz W, Mace GM, Mooers AO, Safi K, Scott O, Steel M, Tucker CM, Pearse WD, Owen NR, Rosindell J (2023) The EDGE2 protocol: Advancing the prioritisation of Evolutionarily Distinct and Globally Endangered species for practical conservation action. *PLOS Biology*, 21:e3001991.

Nogués-Bravo D (2009) Predicting the past distribution of species climatic niches. Predicting the past distribution of species climatic niches. *Global Ecology and Biogeography*, 18:521–531.

Payne JL, Finnegan S (2007) The effect of geographic range on extinction risk during background and mass extinction. *Proceedings of the National Academy of Sciences*, 104:10506–10511.

Purvis A, Gittleman JL, Cowlishaw G, Mace GM (2000) Predicting extinction risk in declining species. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 267:1947–1952.

Taylor C, Cadenhead N, Lindenmayer DB, Wintle BA (2017) Improving the design of a conservation reserve for a critically endangered species. *PLoS ONE*, 12:e0169629.

Thomas CD, Cameron A, Green RE, Bakkenes M, Beaumont LJ, Collingham YC, Erasmus BFN, de Siqueira MF, Grainger A, Hannah L, Hughes L, Huntley B, van Jaarsveld AS, Midgley GF, Miles L, Ortega-Huerta MA, Townsend Peterson A, Phillips OL, Williams SE (2004) Extinction risk from climate change. *Nature* 427:145–148.

See Also

Other target setting methods: spec_absolute_targets(), spec_area_targets(), spec_interp_absolute_targets(), spec_interp_area_targets(), spec_jung_targets(), spec_max_targets(), spec_min_targets(), spec_polak_targets(), spec_pop_size_targets(), spec_relative_targets(), spec_rl_ecosystem_targets(), spec_rl_species_targets(), spec_rodrigues_targets(), spec_rule_targets(), spec_ward_targets(), spec_watson_targets(), spec_wilson_targets()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_complex_pu_raster <- get_sim_complex_pu_raster()
sim_complex_features <- get_sim_complex_features()
sim_complex_historical_features <- get_sim_complex_historical_features()

# calculate the total historical distribution size for each feature.
# note that here we assume that the features in both sim_complex_features
# and sim_complex_historical_features follow the same ordering
historical_distribution_size <- as.numeric(units::set_units(
 units::set_units(
   terra::global(sim_complex_historical_features, "sum", na.rm = TRUE)[[1]] *
   prod(terra::res(sim_complex_historical_features)),
   "m^2"
 ),
 "km^2"
))

# create base problem
p0 <-
  problem(sim_complex_pu_raster, sim_complex_features) %>%
  add_min_set_objective() %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# create problem with targets based on the minimum amount of habitat required
# to ensure that each species has a 95% probability of persistence,
# following Duran et al. (2020)
# a 95% probability of persistence
p1 <-
 p0 %>%
 add_auto_targets(
    method = spec_duran_targets(
     probability_target = 0.95,
     historical_area = historical_distribution_size,
     area_units = "km^2"
   )
 )

# solve problem
```

```
s1 <- solve(p1)

# plot solution
plot(s1, main = "solution based on 95% persistence targets", axes = FALSE)

# create problem with targets based on the minimum amount of habitat required
# to ensure that each species has a particular probability of persistence,
# following Duran et al. (2020)

# simulate a probability of persistence value for each feature
sim_probs <- runif(terra::nlyr(sim_complex_features), 0.1, 0.99)

# now, create problem with these targets
p2 <-
 p0 %>%
 add_auto_targets(
    method = spec_duran_targets(
     probability_target = sim_probs,
     historical_area = historical_distribution_size,
     area_units = "km^2"
   )
 )

# solve problem
s2 <- solve(p2)

# plot solution
plot(s2, main = "solution based on varying targets", axes = FALSE)

## End(Not run)
```

`spec_interp_absolute_targets` *Specify targets based on interpolating absolute thresholds*

Description

Specify targets by interpolating them between thresholds expressed as the same values as the underlying feature data (ignoring any specified feature units). Briefly, this method involves (i) setting target thresholds for rare features to a particular percentage threshold, (ii) setting target thresholds for common features to a particular percentage threshold, and (iii) interpolating target thresholds for features with spatial distributions that range between the those for the rare and common features. Additionally, features can (optionally) have their targets capped at a particular threshold. This method is especially useful for setting targets based on interpolation procedures when features do not have data expressed as an area-based unit of measurement. Note that this function is designed to be used with `add_auto_targets()` and `add_group_targets()`.

Usage

```
spec_interp_absolute_targets(
  rare_absolute_threshold,
  rare_relative_target,
  rare_absolute_target,
  rare_method,
  common_absolute_threshold,
  common_relative_target,
  common_absolute_target,
  common_method,
  cap_absolute_target,
  interp_method
)
```

Arguments

`rare_absolute_threshold`
: `numeric` value indicating the absolute threshold for identifying rare features. This value must be expressed in the same units as the feature data. In particular, features with a total abundance smaller than this value will be considered rare during the target setting calculations.

`rare_relative_target`
: `numeric` value indicating the relative target for rare features. Note that this value must be a proportion between 0 and 1. For example, a value of 0.1 corresponds to 10%.

`rare_absolute_target`
: `numeric` value denoting the absolute target for rare features. This value must be expressed in the same units as the feature data. To avoid setting an absolute target for rare features, a missing (`NA`) value can be specified.

`rare_method`
: `character` value indicating how the target for rare features should be calculated. Available options include `"min"` and `"max"`. For example, a value of `"max"` means that the target for a rare features is calculated as the maximum based on `rare_relative_target` and `rare_absolute_target`. Note that `rare_method` will have no effect on the target calculations if `rare_absolute_target` is a missing (`NA`) value.

`common_absolute_threshold`
: `numeric` value indicating the absolute threshold for identifying common features. This value must be expressed in the same units as the feature data. In particular, features with a total abundance greater than this value will be considered common during the target setting calculations.

`common_relative_target`
: `numeric` value denoting the relative target for common features. Note that this value must be a proportion between 0 and 1. For example, a value of 0.1 corresponds to 10%.

`common_absolute_target`
: `numeric` value denoting the absolute target for common features. This value must be expressed in the same units as the feature data. To avoid setting an absolute target for common features, a missing (`NA`) value can be specified.

`common_method` `character` value indicating how the target for common features should be calculated. Available options include `"min"` and `"max"`. For example, a value of `"max"` means that the target for a common feature is calculated as the maximum based on `common_relative_target` and `common_absolute_target`. Note that `common_method` will have no effect on the target calculations if `common_absolute_target` is a missing (`NA`) value.

`cap_absolute_target`
`numeric` value denoting the absolute target cap. This value must be expressed in the same units as the feature data. In particular, all targets are clamped to this value during target setting calculations. To avoid setting a target cap, a missing (`NA`) value can be specified.

`interp_method` `character` value denoting the interpolation method. Available options include `"linear"` for linear interpolation and `"log10"` for log-linear interpolation.

Details

This method has been applied to set target thresholds at global and national scales (e.g., Butchart *et al.* 2015; Rodrigues *et al.* 2004; Polak *et al.* 2015). It is based on the rationale that species with a smaller geographic distribution are at a greater risk of extinction, and so require a larger percentage of their geographic distribution to be represented by a prioritization (Rodrigues *et al.* 2004). When using this method in a planning exercise, it is important to ensure that the threshold parameters reflect the stakeholder objectives. Additionally, the threshold parameters may need to set according to the spatial extent of the planning region.

Value

An object (`TargetMethod`) for specifying targets that can be used with `add_auto_targets()` and `add_group_targets()` to add targets to a `problem()`.

Mathematical formulation

This method provides a flexible approach for setting target thresholds based on an interpolation procedure and the feature data. To express this mathematically, we will define the following terminology. Let f denote the total abundance of a feature, a the threshold for identifying rare features (per `rare_absolute_threshold`), b the relative targets for rare features (per `rare_relative_target`), c the absolute targets for rare features (per `rare_absolute_target`), $d()$ the function for calculating targets for rare features as a maximum or minimum value (per `rare_method`), e the threshold for identifying common features (per `common_absolute_threshold`), g the relative targets for common features (per `common_relative_target`), h the absolute targets for common features (per `common_absolute_target`), $i()$ the method for calculating targets for common features as a maximum or minimum value (per `common_method`), j the target cap (per `cap_absolute_target`), and $k()$ the interpolation method for features with a spatial distribution that is larger than a rare features and smaller than a common feature (per `interp_method`). In particular, $k()$ is either a linear or log-linear interpolation procedure based on the thresholds for identifying rare and common features as well as the relative targets for rare and common features. Given this terminology, the target threshold (t) for the feature is calculated as follows.

- If $f < a$, then $t = min(d(c, b \times f), j)$.
- If $f > e$, then $t = min(i(h, g \times f), j)$.

- If $a \leq f \leq e$, then $t = min(k(f, a, b, e, g), j)$.

References

Butchart SHM, Clarke M, Smith RJ, Sykes RE, Scharlemann JPW, Harfoot M, Buchanan GM, Angulo A, Balmford A, Bertzky B, Brooks TM, Carpenter KE, Comeros-Raynal MT, Cornell J, Ficetola GF, Fishpool LDC, Fuller RA, Geldmann J, Harwell H, Hilton-Taylor C, Hoffmann M, Joolia A, Joppa L, Kingston N, May I, Milam A, Polidoro B, Ralph G, Richman N, Rondinini C, Segan DB, Skolnik B, Spalding MD, Stuart SN, Symes A, Taylor J, Visconti P, Watson JEM, Wood L, Burgess ND (2015) Shortfalls and solutions for meeting national and global conservation area targets. *Conservation Letters*, 8: 329–337.

Polak T, Watson JEM, Fuller RA, Joseph LN, Martin TG, Possingham HP, Venter O, Carwardine J (2015) Efficient expansion of global protected areas requires simultaneous planning for species and ecosystems. *Royal Society Open Science*, 2: 150107.

Rodrigues ASL, Akçakaya HR, Andelman SJ, Bakarr MI, Boitani L, Brooks TM, Chanson JS, Fishpool LDC, Da Fonseca GAB, Gaston KJ, Hoffmann M, Marquet PA, Pilgrim JD, Pressey RL, Schipper J, Sechrest W, Stuart SN, Underhill LG, Waller RW, Watts MEJ, Yan X (2004) Global gap analysis: priority regions for expanding the global protected-area network. *BioScience*, 54: 1092–1100.

See Also

Other target setting methods: spec_absolute_targets(), spec_area_targets(), spec_duran_targets(), spec_interp_area_targets(), spec_jung_targets(), spec_max_targets(), spec_min_targets(), spec_polak_targets(), spec_pop_size_targets(), spec_relative_targets(), spec_rl_ecosystem_targets(), spec_rl_species_targets(), spec_rodrigues_targets(), spec_rule_targets(), spec_ward_targets(), spec_watson_targets(), spec_wilson_targets()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()

# this function sets targets based on the total abundance of the features
# (i.e., sum of planning unit values for the feature) and does not
# consider the spatial area covered by the planning units

# display the total abundance of the features
print(terra::global(get_sim_features(), "sum", na.rm = TRUE))

# create problem with interpolated targets.
# here, targets will be set as 70% for features with a total abundance
# (i.e., sum of planning unit values for the feature) smaller than 50,
# 20% for features with at total abundance greater than 70,
# linearly interpolated for features with an intermediate range size,
```

```
# and capped at a total abundance of 100
p1 <-
  problem(sim_pu_raster, sim_features) %>%
  add_min_set_objective() %>%
  add_auto_targets(
    method = spec_interp_absolute_targets(
      rare_absolute_threshold = 50,
      rare_relative_target = 0.7,
      rare_absolute_target = NA,            # not used
      rare_method = "max",                  # not used
      common_absolute_threshold = 70,
      common_relative_target = 0.2,
      common_absolute_target = NA,          # not used
      common_method = "max",                # not used
      cap_absolute_target = 100,
      interp_method = "linear"
    )
  ) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s1 <- solve(p1)

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

## End(Not run)
```

`spec_interp_area_targets` *Specify targets based on interpolating area-based thresholds*

Description

Specify targets by interpolating them between area-based thresholds. Briefly, this method involves (i) setting target thresholds for rare features to a particular percentage threshold, (ii) setting target thresholds for common features to a particular percentage threshold, and (iii) interpolating target thresholds for features with spatial distributions that range between the those for the rare and common features. Additionally, features can (optionally) have their targets capped at a particular threshold. This method is especially useful for setting targets based on interpolation procedures when features have data expressed as an area-based unit of measurement (e.g., km^2). Note that this function is designed to be used with `add_auto_targets()` and `add_group_targets()`.

Usage

```
spec_interp_area_targets(
  rare_area_threshold,
  rare_relative_target,
```

```
  rare_area_target,
  rare_method,
  common_area_threshold,
  common_relative_target,
  common_area_target,
  common_method,
  cap_area_target,
  interp_method,
  area_units
)
```

Arguments

`rare_area_threshold`
: `numeric` value indicating the threshold area for identifying rare features. This value must be expressed in the same units as the feature data. In particular, features with a total spatial extent smaller than this value will be considered rare during the target setting calculations.

`rare_relative_target`
: `numeric` value indicating the relative target for rare features. Note that this value must be a proportion between 0 and 1. For example, a value of 0.1 corresponds to 10%.

`rare_area_target`
: `numeric` value denoting the area-based target for rare features. This value must be expressed in the same units as `area_units`. To avoid setting an area-based target for rare features, a missing (`NA`) value can be specified.

`rare_method`
: `character` value indicating how the target for rare features should be calculated. Available options include `"min"` and `"max"`. For example, a value of `"max"` means that the target for a rare features is calculated as the maximum based on `rare_relative_target` and `rare_area_threshold`. Note that `rare_method` will have no effect on the target calculations if `rare_area_target` is a missing (`NA`) value.

`common_area_threshold`
: `numeric` value indicating the threshold area for identifying common features. This value must be expressed in the same units as `area_units`. In particular, features with a total spatial extent greater than this value will be considered common during the target setting calculations.

`common_relative_target`
: `numeric` value denoting the relative target for common features. Note that this value must be a proportion between 0 and 1. For example, a value of 0.1 corresponds to 10%.

`common_area_target`
: `numeric` value denoting the area-based target for common features. This value must be expressed in the same units as `area_units`. To avoid setting an area-based target for common features, a missing (`NA`) value can be specified.

`common_method`
: `character` value indicating how the target for common features should be calculated. Available options include `"min"` and `"max"`. For example, a value of `"max"` means that the target for a common feature is calculated as the maximum

based on common_relative_target and common_area_threshold. Note that common_method will have no effect on the target calculations if common_area_target is a missing (NA) value.

cap_area_target
: numeric value denoting the area-based target cap. This value must be expressed in the same units as area_units. In particular, all targets are clamped to this value during target setting calculations. To avoid setting a target cap, a missing (NA) value can be specified.

interp_method
: character value denoting the interpolation method. Available options include "linear" for linear interpolation and "log10" for log-linear interpolation.

area_units
: character value denoting the unit of measurement for the area-based arguments (e.g., "km^2", "ha", "acres"‘).

Details

This method has been applied to set target thresholds at global and national scales (e.g., Butchart *et al.* 2015; Rodrigues *et al.* 2004; Polak *et al.* 2015). It is based on the rationale that species with a smaller geographic distribution are at a greater risk of extinction, and so require a larger percentage of their geographic distribution to be represented by a prioritization (Rodrigues *et al.* 2004). When using this method in a planning exercise, it is important to ensure that the threshold parameters reflect the stakeholder objectives. Additionally, the threshold parameters may need to set according to the spatial extent of the planning region.

Value

An object (TargetMethod) for specifying targets that can be used with add_auto_targets() and add_group_targets() to add targets to a problem().

Mathematical formulation

This method provides a flexible approach for setting target thresholds based on an interpolation procedure and the spatial extent of the features. To express this mathematically, we will define the following terminology. Let f denote the total spatial extent of a feature (e.g., geographic range size), a the threshold for identifying rare features (per rare_area_threshold and area_units), b the relative targets for rare features (per rare_relative_target), c the area-based targets for rare features (per rare_area_target and area_units), $d()$ the function for calculating targets for rare features as a maximum or minimum value (per rare_method), e the threshold for identifying common features (per common_area_threshold and area_units), g the relative targets for common features (per common_relative_target), h the area-based targets for common features (per common_area_target and area_units), $i()$ the method for calculating targets for common features as a maximum or minimum value (per common_method), and j the target cap (per cap_area_target and area_units), and $k()$ the interpolation method for features with a spatial distribution that is larger than a rare features and smaller than a common feature (per interp_method). In particular, $k()$ is either a linear or log-linear interpolation procedure based on the thresholds for identifying rare and common features as well as the relative targets for rare and common features. Given this terminology, the target threshold (t) for the feature is calculated as follows.

- If $f < a$, then $t = min(d(c, b \times f), j)$.
- If $f > e$, then $t = min(i(h, g \times f), j)$.

- If $a \leq f \leq e$, then $t = min(k(f, a, b, e, g), j)$.

Data calculations

This function involves calculating targets based on the spatial extent of the features in `x`. Although it can be readily applied to `problem()` objects that have the feature data provided as a `terra::rast()` object, you will need to specify the spatial units for the features when initializing the `problem()` objects if the feature data are provided in a different format. In particular, if the feature data are provided as a `data.frame` or `character` vector, then you will need to specify an argument to `feature_units` when using the `problem()` function. See the Examples section of the documentation for `add_auto_targets()` for a demonstration of specifying the spatial units for features.

References

Butchart SHM, Clarke M, Smith RJ, Sykes RE, Scharlemann JPW, Harfoot M, Buchanan GM, Angulo A, Balmford A, Bertzky B, Brooks TM, Carpenter KE, Comeros-Raynal MT, Cornell J, Ficetola GF, Fishpool LDC, Fuller RA, Geldmann J, Harwell H, Hilton-Taylor C, Hoffmann M, Joolia A, Joppa L, Kingston N, May I, Milam A, Polidoro B, Ralph G, Richman N, Rondinini C, Segan DB, Skolnik B, Spalding MD, Stuart SN, Symes A, Taylor J, Visconti P, Watson JEM, Wood L, Burgess ND (2015) Shortfalls and solutions for meeting national and global conservation area targets. *Conservation Letters*, 8: 329–337.

Polak T, Watson JEM, Fuller RA, Joseph LN, Martin TG, Possingham HP, Venter O, Carwardine J (2015) Efficient expansion of global protected areas requires simultaneous planning for species and ecosystems. *Royal Society Open Science*, 2: 150107.

Rodrigues ASL, Akçakaya HR, Andelman SJ, Bakarr MI, Boitani L, Brooks TM, Chanson JS, Fishpool LDC, Da Fonseca GAB, Gaston KJ, Hoffmann M, Marquet PA, Pilgrim JD, Pressey RL, Schipper J, Sechrest W, Stuart SN, Underhill LG, Waller RW, Watts MEJ, Yan X (2004) Global gap analysis: priority regions for expanding the global protected-area network. *BioScience*, 54: 1092–1100.

See Also

Other target setting methods: `spec_absolute_targets()`, `spec_area_targets()`, `spec_duran_targets()`, `spec_interp_absolute_targets()`, `spec_jung_targets()`, `spec_max_targets()`, `spec_min_targets()`, `spec_polak_targets()`, `spec_pop_size_targets()`, `spec_relative_targets()`, `spec_rl_ecosystem_targets()`, `spec_rl_species_targets()`, `spec_rodrigues_targets()`, `spec_rule_targets()`, `spec_ward_targets()`, `spec_watson_targets()`, `spec_wilson_targets()`

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_complex_pu_raster <- get_sim_complex_pu_raster()
sim_complex_features <- get_sim_complex_features()

# create problem with interpolated targets.
# here, targets will be set as 100% for features smaller than 1000 km^2
```

```
# in size, 10% for features greater than 250,000 km^2 in size,
# log-linearly interpolated for features with an intermediate range size,
# and capped at 1,000,000 km^2
p1 <-
  problem(sim_complex_pu_raster, sim_complex_features) %>%
  add_min_set_objective() %>%
  add_auto_targets(
    method = spec_interp_area_targets(
     rare_area_threshold = 1000,
     rare_relative_target = 1,
     rare_area_target = NA,            # not used
     rare_method = "max",              # not used
     common_area_threshold = 250000,
     common_relative_target = 0.1,
     common_area_target = NA,          # not used
     common_method = "max",            # not used
     cap_area_target = 1000000,
     interp_method = "log10",
     area_units = "km^2"
    )
  ) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s1 <- solve(p1)

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

## End(Not run)
```

`spec_jung_targets` *Specify targets following Jung* et al. *(2021)*

Description

Specify targets based on the methodology outlined by Jung *et al.* (2021). Briefly, this method involves setting targets based the criteria for recognizing Vulnerable species by the International Union for the Conservation of Nature (IUCN) Red List of Threatened Species (IUCN 2025). To help prevent widespread features from obscuring priorities, targets are capped following Butchart *et al.* (2015). This method was designed for global-scale prioritizations. Note that this function is designed to be used with `add_auto_targets()` and `add_group_targets()`.

Usage

```
spec_jung_targets(
  status = "VU",
  prop_uplift = 0.1,
```

```
  cap_area_target = 1e+06,
  area_units = "km^2"
)
```

Arguments

status	character value denoting the IUCN Red List threat status used for target setting. Available options include "CR" (Critically Endangered) , "EN" (Endangered), and "VU" (Vulnerable). Defaults to "VU".
prop_uplift	numeric value denoting the percentage uplift as a proportion. Defaults to 0.1 (i.e., 10%).
cap_area_target	numeric value denoting the area-based target cap. To avoid setting a target cap, a missing (NA) value can be specified. Defaults to 1000000 (i.e., 1,000,000 km^2).
area_units	character value denoting the unit of measurement for the area-based arguments. Defaults to "km^2" (i.e., km^2).

Details

This target setting method was designed to protect species in global-scale prioritizations (Jung *et al.* 2021). Although it has been successfully applied to multiple global-scale prioritizations (e.g., Mogg *et al.* 2019; Fastré *et al.* 2021), it may fail to identify meaningful priorities for prioritizations conducted at smaller geographic scales (e.g., national, state-level, or county scales). For example, if this target setting method is applied to smaller geographic scales, then the resulting prioritizations may select an overly large percentage of the study area, or be biased towards over-representing common and widespread species. This is because the target thresholds were developed based on criteria for promoting the long-term persistence of entire species. As such, if you are working at smaller scales that do not fully cover the entire spatial distribution of the study species, then you may need to rescale these targets (e.g., based on the proportion of the species' distribution found within the study area) or consider an alternative target setting method. Please note that this function is provided as convenient method to set targets for problems with a single management zone, and cannot be used for those with multiple management zones.

Value

An object (TargetMethod) for specifying targets that can be used with add_auto_targets() and add_group_targets() to add targets to a problem().

Data calculations

This function involves calculating targets based on the spatial extent of the features in x. Although it can be readily applied to problem() objects that have the feature data provided as a terra::rast() object, you will need to specify the spatial units for the features when initializing the problem() objects if the feature data are provided in a different format. In particular, if the feature data are provided as a data.frame or character vector, then you will need to specify an argument to feature_units when using the problem() function. See the Examples section of the documentation for add_auto_targets() for a demonstration of specifying the spatial units for features.

Mathematical formulation

This method involves setting target thresholds based on assessment criteria from the International Union for the Conservation of Nature (IUCN) Red List of Threatened Species (IUCN 2025). In particular, it considers criteria related to the size of a species' spatial distribution (i.e., Criterion B2) and population size reduction (i.e., Criterion A2) and applies a percentage-based uplift to them. By default, it considers criteria for the Vulnerable threat status with a 10% uplift and involves setting the target threshold for a species as 2,200 km^2 or 80% of its spatial distribution (which ever value is larger). Additionally, following Butchart *et al.* (2015), a cap of 1,000,000 km^2 is applied to target thresholds.

References

Butchart SHM, Clarke M, Smith RJ, Sykes RE, Scharlemann JPW, Harfoot M, Buchanan GM, Angulo A, Balmford A, Bertzky B, Brooks TM, Carpenter KE, Comeros-Raynal MT, Cornell J, Ficetola GF, Fishpool LDC, Fuller RA, Geldmann J, Harwell H, Hilton-Taylor C, Hoffmann M, Joolia A, Joppa L, Kingston N, May I, Milam A, Polidoro B, Ralph G, Richman N, Rondinini C, Segan DB, Skolnik B, Spalding MD, Stuart SN, Symes A, Taylor J, Visconti P, Watson JEM, Wood L, Burgess ND (2015) Shortfalls and solutions for meeting national and global conservation area targets. *Conservation Letters*, 8: 329–337.

Fastré C, van Zeist W-J, Watson JEM, Visconti P (2021) Integrated spatial planning for biodiversity conservation and food production. *One Earth*, 4:1635–1644.

IUCN (2025) The IUCN Red List of Threatened Species. Version 2025-1. Available at https://www.iucnredlist.org. Accessed on 23 July 2025.

Jung M, Arnell A, de Lamo X, García-Rangel S, Lewis M, Mark J, Merow C, Miles L, Ondo I, Pironon S, Ravilious C, Rivers M, Schepaschenko D, Tallowin O, van Soesbergen A, Govaerts R, Boyle BL, Enquist BJ, Feng X, Gallagher R, Maitner B, Meiri S, Mulligan M, Ofer G, Roll U, Hanson JO, Jetz W, Di Marco M, McGowan J, Rinnan DS, Sachs JD, Lesiv M, Adams VM, Andrew SC, Burger JR, Hannah L, Marquet PA, McCarthy JK, Morueta-Holme N, Newman EA, Park DS, Roehrdanz PR, Svenning J-C, Violle C, Wieringa JJ, Wynne G, Fritz S, Strassburg BBN, Obersteiner M, Kapos V, Burgess N, Schmidt- Traub G, Visconti P (2021) Areas of global importance for conserving terrestrial biodiversity, carbon and water. *Nature Ecology and Evolution*, 5:1499–1509.

Mogg S, Fastre C, Jung M, Visconti P (2019) Targeted expansion of Protected Areas to maximise the persistence of terrestrial mammals. *Preprint at bioxriv*, doi:10.1101/608992.

See Also

See targets for an overview of all functions for adding targets.

Other target setting methods: `spec_absolute_targets()`, `spec_area_targets()`, `spec_duran_targets()`, `spec_interp_absolute_targets()`, `spec_interp_area_targets()`, `spec_max_targets()`, `spec_min_targets()`, `spec_polak_targets()`, `spec_pop_size_targets()`, `spec_relative_targets()`, `spec_rl_ecosystem_targets()`, `spec_rl_species_targets()`, `spec_rodrigues_targets()`, `spec_rule_targets()`, `spec_ward_targets()`, `spec_watson_targets()`, `spec_wilson_targets()`

Examples

```
## Not run:
# set seed for reproducibility
```

```
set.seed(500)

# load data
sim_complex_pu_raster <- get_sim_complex_pu_raster()
sim_complex_features <- get_sim_complex_features()

# create problem with Jung et al. (2021) targets
p1 <-
  problem(sim_complex_pu_raster, sim_complex_features) %>%
  add_min_set_objective() %>%
  add_auto_targets(method = spec_jung_targets()) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s1 <- solve(p1)

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

## End(Not run)
```

`spec_max_targets` *Specify targets based on maxima*

Description

Specify targets that are calculated based on the maximum of one or more target setting methods.

Usage

```
spec_max_targets(x, ...)
```

Arguments

`x`	An object specifying a target setting method.
`...`	Additional objects specifying target setting methods.

Value

An object (`TargetMethod`) for specifying targets.

Data calculations

This function involves calculating targets based on the spatial extent of the features in `x`. Although it can be readily applied to `problem()` objects that have the feature data provided as a `terra::rast()` object, you will need to specify the spatial units for the features when initializing the `problem()` objects if the feature data are provided in a different format. In particular, if the feature data are provided as a `data.frame` or `character` vector, then you will need to specify an argument to `feature_units` when using the `problem()` function. See the Examples section of the documentation for `add_auto_targets()` for a demonstration of specifying the spatial units for features.

See Also

Other target setting methods: spec_absolute_targets(), spec_area_targets(), spec_duran_targets(), spec_interp_absolute_targets(), spec_interp_area_targets(), spec_jung_targets(), spec_min_targets(), spec_polak_targets(), spec_pop_size_targets(), spec_relative_targets(), spec_rl_ecosystem_targets(), spec_rl_species_targets(), spec_rodrigues_targets(), spec_rule_targets(), spec_ward_targets(), spec_watson_targets(), spec_wilson_targets()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_complex_pu_raster <- get_sim_complex_pu_raster()
sim_complex_features <- get_sim_complex_features()

# create base problem
p0 <-
  problem(sim_complex_pu_raster, sim_complex_features) %>%
  add_min_set_objective() %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# create problem with 20% targets
p1 <-
  p0 %>%
  add_auto_targets(method = spec_relative_targets(0.2))

# create problem with Jung et al. (2021) targets
p2 <-
  p0 %>%
  add_auto_targets(method = spec_jung_targets())

# create problem with Polak et al. (2015) targets
p3 <-
  p0 %>%
  add_auto_targets(method = spec_polak_targets())

# create problem with targets based on the maximum of 20% targets,
# Jung et al. (2021) targets, and Polak et al. (2015) targets
# for each feature (separately)
p4 <-
  p0 %>%
  add_auto_targets(
    method = spec_max_targets(
      spec_relative_targets(0.2),
      spec_jung_targets(),
      spec_polak_targets()
    )
  )
```

```
# solve problems
s <- c(solve(p1), solve(p2), solve(p3), solve(p4))
names(s) <- c("20% targets", "Jung targets", "Polak targets", "max targets")

# plot solutions
plot(s, axes = FALSE)

## End(Not run)
```

`spec_min_targets` *Specify targets based on minima*

Description

Specify targets that are calculated based on the minimum of one or more target setting methods.

Usage

```
spec_min_targets(x, ...)
```

Arguments

`x`	An object specifying a target setting method.
`...`	Additional objects specifying target setting methods.

Value

An object (`TargetMethod`) for specifying targets.

Data calculations

This function involves calculating targets based on the spatial extent of the features in `x`. Although it can be readily applied to `problem()` objects that have the feature data provided as a `terra::rast()` object, you will need to specify the spatial units for the features when initializing the `problem()` objects if the feature data are provided in a different format. In particular, if the feature data are provided as a `data.frame` or `character` vector, then you will need to specify an argument to `feature_units` when using the `problem()` function. See the Examples section of the documentation for `add_auto_targets()` for a demonstration of specifying the spatial units for features.

See Also

Other target setting methods: `spec_absolute_targets()`, `spec_area_targets()`, `spec_duran_targets()`, `spec_interp_absolute_targets()`, `spec_interp_area_targets()`, `spec_jung_targets()`, `spec_max_targets()`, `spec_polak_targets()`, `spec_pop_size_targets()`, `spec_relative_targets()`, `spec_rl_ecosystem_targets()`, `spec_rl_species_targets()`, `spec_rodrigues_targets()`, `spec_rule_targets()`, `spec_ward_targets()`, `spec_watson_targets()`, `spec_wilson_targets()`

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_complex_pu_raster <- get_sim_complex_pu_raster()
sim_complex_features <- get_sim_complex_features()

# create base problem
p0 <-
  problem(sim_complex_pu_raster, sim_complex_features) %>%
  add_min_set_objective() %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# create problem with 20% targets
p1 <-
  p0 %>%
  add_auto_targets(method = spec_relative_targets(0.2))

# create problem with Jung et al. (2021) targets
p2 <-
  p0 %>%
  add_auto_targets(method = spec_jung_targets())

# create problem with Polak et al. (2015) targets
p3 <-
  p0 %>%
  add_auto_targets(method = spec_polak_targets())

# create problem with targets based on the minimum of 20% targets,
# Jung et al. (2021) targets, and Polak et al. (2015) targets
# for each feature (separately)
p4 <-
  p0 %>%
  add_auto_targets(
    method = spec_min_targets(
      spec_relative_targets(0.2),
      spec_jung_targets(),
      spec_polak_targets()
    )
  )

# solve problems
s <- c(solve(p1), solve(p2), solve(p3), solve(p4))
names(s) <- c("20% targets", "Jung targets", "Polak targets", "min targets")

# plot solutions
plot(s, axes = FALSE)

## End(Not run)
```

spec_polak_targets *Specify targets following Polak* et al. *(2015)*

Description

Specify targets based on the methodology outlined by Polak *et al.* (2015). Briefly, this method involves setting targets based on linear interpolation methods. To help prevent widespread features from obscuring priorities, targets are capped following Butchart *et al.* (2015). Note that this function is designed to be used with `add_auto_targets()` and `add_group_targets()`.

Usage

```
spec_polak_targets(
  rare_area_threshold = 1000,
  rare_relative_target = 1,
  common_area_threshold = 10000,
  common_relative_target = 0.1,
  cap_area_target = 1e+06,
  area_units = "km^2"
)
```

Arguments

`rare_area_threshold`
: `numeric` value indicating the threshold area for rare identifying rare features. Defaults to 1000 (i.e., 1000 km^2).

`rare_relative_target`
: `numeric` value denoting the relative target for features with a spatial distribution that is smaller than `rare_area_threshold`. Note that this value must be a proportion between 0 and 1. Defaults to 1 (i.e., 100%).

`common_area_threshold`
: `numeric` value indicating the threshold area for identifying common features. Defaults to 10000 (i.e., 10,000 km^2).

`common_relative_target`
: `numeric` value denoting the relative target for features with a spatial distribution that is greater than `common_area_threshold`. Defaults to 0.1 (i.e., 10%). Since this default value is based on historical levels of global protected area coverage, it may be appropriate to set this value based on current levels of protected area coverage (e.g., 17.6% for terrestrial and 8.4% for marine systems globally; UNEP-WCMC and IUCN 2025).

`cap_area_target`
: `numeric` value denoting the area-based target cap. To avoid setting a target cap, a missing (`NA`) value can be specified. Defaults to 1000000 (i.e., 1,000,000 km^2).

`area_units`
: `character` value denoting the unit of measurement for the area-based arguments. Defaults to `"km^2"` (i.e., km^2).

Details

This target setting method was designed to protect species in national- scale prioritizations (Polak *et al.* 2015). Although it has been successfully applied to to national-scales (e.g., Polak *et al.* 2016; Clements *et al.* 2018;), it may fail to identify meaningful priorities for prioritizations conducted at smaller or larger geographic scales (e.g., local or global scales). For example, if this method is applied to smaller geographic scales, then the resulting prioritizations may select an overly large percentage of the study area, or be biased towards over-representing common and widespread species. This is because the thresholds for defining rare and common features (i.e., `rare_area_threshold` and `common_area_threshold`) were originally developed based on criteria for national-scales. As such, if you working at a different scale, you may need to calibrate these thresholds based on the spatial extent of the planning region. Please note that this function is provided as convenient method to set targets for problems with a single management zone, and cannot be used for those with multiple management zones.

Value

An object (`TargetMethod`) for specifying targets that can be used with `add_auto_targets()` and `add_group_targets()` to add targets to a `problem()`.

Mathematical formulation

This method involves setting target thresholds based on the spatial extent of the features. By default, this method identifies rare features as those with a spatial distribution smaller than 1,000 km^2 (per `rare_area_threshold` and `area_units`) and common features as those with a spatial distribution larger than 10,000 km^2 (per `common_area_threshold` and `area_units`). Given this, rare features are assigned a target threshold of 100% (per `rare_relative_target`), common features are assigned a target threshold of 10% (per `common_relative_target`), and features with a spatial distribution that is between the area-based thresholds used to identify rare and common features are assigned a target threshold through linear interpolation. Additionally, following Butchart *et al.* (2015), a cap of 1,000,000 km^2 is applied to target thresholds (per `cap_area_threshold` and `area_units`).

Data calculations

This function involves calculating targets based on the spatial extent of the features in `x`. Although it can be readily applied to `problem()` objects that have the feature data provided as a `terra::rast()` object, you will need to specify the spatial units for the features when initializing the `problem()` objects if the feature data are provided in a different format. In particular, if the feature data are provided as a `data.frame` or `character` vector, then you will need to specify an argument to `feature_units` when using the `problem()` function. See the Examples section of the documentation for `add_auto_targets()` for a demonstration of specifying the spatial units for features.

References

Butchart SHM, Clarke M, Smith RJ, Sykes RE, Scharlemann JPW, Harfoot M, Buchanan GM, Angulo A, Balmford A, Bertzky B, Brooks TM, Carpenter KE, Comeros-Raynal MT, Cornell J, Ficetola GF, Fishpool LDC, Fuller RA, Geldmann J, Harwell H, Hilton-Taylor C, Hoffmann M, Joolia A, Joppa L, Kingston N, May I, Milam A, Polidoro B, Ralph G, Richman N, Rondinini C, Segan DB, Skolnik B, Spalding MD, Stuart SN, Symes A, Taylor J, Visconti P, Watson JEM, Wood

L, Burgess ND (2015) Shortfalls and solutions for meeting national and global conservation area targets. *Conservation Letters*, 8: 329–337.

Clements HS, Kearney SG, Cook CN (2018) Moving from representation to persistence: The capacity of Australia's National Reserve System to support viable populations of mammals. *Diversity and Distributions*, 24: 1231–1241.

Polak T, Watson JEM, Fuller RA, Joseph LN, Martin TG, Possingham HP, Venter O, Carwardine J (2015) Efficient expansion of global protected areas requires simultaneous planning for species and ecosystems. *Royal Society Open Science*, 2: 150107.

Polak T, Watson JEM, Bennett JR, Possingham HP, Fuller RA, Carwardine J (2016) Balancing ecosystem and threatened species representation in protected areas and implications for nations achieving global conservation goals. *Conservation Letters*, 9:438–445.

UNEP-WCMC and IUCN (2025) Protected Planet Report 2024. Cambridge, UK: UNEP-WCMC and IUCN. Available at <www.protectedplanet.net>.

See Also

Other target setting methods: spec_absolute_targets(), spec_area_targets(), spec_duran_targets(), spec_interp_absolute_targets(), spec_interp_area_targets(), spec_jung_targets(), spec_max_targets(), spec_min_targets(), spec_pop_size_targets(), spec_relative_targets(), spec_rl_ecosystem_targets(), spec_rl_species_targets(), spec_rodrigues_targets(), spec_rule_targets(), spec_ward_targets(), spec_watson_targets(), spec_wilson_targets()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_complex_pu_raster <- get_sim_complex_pu_raster()
sim_complex_features <- get_sim_complex_features()

# create problem with Polak et al. (2015) targets
p1 <-
  problem(sim_complex_pu_raster, sim_complex_features) %>%
  add_min_set_objective() %>%
  add_auto_targets(method = spec_polak_targets()) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s1 <- solve(p1)

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

## End(Not run)
```

spec_pop_size_targets *Specify targets based on population size*

Description

Specify targets based on the minimum number of individuals for each feature. Briefly, this method involves using population density data to set a target threshold for the minimum amount of habitat required to safeguard a particular number of individuals. To help prevent widespread features from obscuring priorities, targets are capped following Butchart *et al.* (2015). Note that this function is designed to be used with `add_auto_targets()` and `add_group_targets()`.

Usage

```
spec_pop_size_targets(
  pop_size_targets,
  pop_density,
  density_units,
  cap_area_target = 1e+06,
  area_units = "km^2"
)
```

Arguments

`pop_size_targets`
: `numeric` vector that specifies the minimum population size required for each feature. If a single `numeric` value is specified, then all features are assigned targets based on the same population size.

`pop_density`
: `numeric` vector that specifies the population density for each feature. If a single `numeric` value is specified, then all features are assigned targets assuming the same population density. See Population density section for more details.

`density_units`
: `character` vector that specifies the area-based units for the population density values. For example, units can be used to express that population densities are in terms of individuals per hectare (`"ha"`), acre (`"acre"`), or km^2 (`"km^2"`). If a single `character` value is specified, then all features are assigned targets assuming that population density values are in the same units. See Population density section for more details.

`cap_area_target`
: `numeric` value denoting the area-based target cap. To avoid setting a target cap, a missing (`NA`) value can be specified. Defaults to 1000000 (i.e., 1,000,000 km^2).

`area_units`
: `character` value denoting the unit of measurement for the area-based arguments. Defaults to `"km^2"` (i.e., km^2).

Details

This target setting method can be used to set targets for species based on population size thresholds. Many different population size thresholds – and methods for calculating such thresholds – have been proposed for guiding conservation decisions (Di Marco *et al.* 2016). For example, previous work has suggested that the number of individuals for a population should not fall below 50 individuals to avoid inbreeding depression, and 500 individuals to reduce genetic drift (reviewed by Jamieson and Allendorf 2012). Also, the Red List of Threatened Species by the International Union for the Conservation of Nature has criteria related to population size, where a species with fewer than 250, 2,500, or 10,000 individuals are recognized as Critically Endangered, Endangered, or Vulnerable (respectively) (IUCN 2025). Additionally, the SAFE index (Clements *et al.* 2011) considers species with fewer than 5,000 individuals to be threatened by extinction (based on Brook *et al.* 2006; Traill *et al.* 2007, 2010). Furthermore, Hilbers *et al.* (2017) and Wolff *et al.* (2023) developed methodologies for estimating species-specific population sizes for protection based on population growth rates.

Value

An object (`TargetMethod`) for specifying targets that can be used with `add_auto_targets()` and `add_group_targets()` to add targets to a `problem()`.

Population density

This method requires population density data expressed as the number of individuals per unit area. For example, if a species has 200 individuals per hectare, then this can be specified with `pop_density = 200` and `density_units = "ha"`. Alternatively, if a species has a population density where one individual occurs every 10 km^2, then this can be specified with `pop_density = 0.1` and `density_units = "km^2"`. Also, note that population density is assumed to scale linearly with the values in the feature data. For example, if a planning unit contains 5 km^2 of habitat for a feature, `pop_density = 200`, and `density_units = "km^2"`, then the calculations assume that the planning unit contains 100 individuals for the species. Although the package does not provide the population density data required to apply this target setting method, such data can be obtained from published databases (e.g., Santini *et al.* 2022, 2023, 2024; Witting *et al.* 2024).

Mathematical formulation

This method involves setting target thresholds based on the amount of habitat required to safeguard a pre-specified number of individuals. To express this mathematically, we will define the following terminology. Let f denote the total abundance of a feature (i.e., geographic range size expressed as km^2), n denote the minimum number of individuals that should ideally be represented (per `pop_size_targets`), and d population density of the feature (i.e., number of individuals per km^2, per `pop_density` and `density_units`), and j the target cap (expressed as km^2, per `cap_area_target` and `area_units`). Given this terminology, the target threshold (t) for the feature is calculated as follows.

$$t = min(f, j, n \times d)$$

Data calculations

This function involves calculating targets based on the spatial extent of the features in `x`. Although it can be readily applied to `problem()` objects that have the feature data provided as a `terra::rast()`

object, you will need to specify the spatial units for the features when initializing the `problem()` objects if the feature data are provided in a different format. In particular, if the feature data are provided as a `data.frame` or `character` vector, then you will need to specify an argument to `feature_units` when using the `problem()` function. See the Examples section of the documentation for `add_auto_targets()` for a demonstration of specifying the spatial units for features.

References

Butchart SHM, Clarke M, Smith RJ, Sykes RE, Scharlemann JPW, Harfoot M, Buchanan GM, Angulo A, Balmford A, Bertzky B, Brooks TM, Carpenter KE, Comeros-Raynal MT, Cornell J, Ficetola GF, Fishpool LDC, Fuller RA, Geldmann J, Harwell H, Hilton-Taylor C, Hoffmann M, Joolia A, Joppa L, Kingston N, May I, Milam A, Polidoro B, Ralph G, Richman N, Rondinini C, Segan DB, Skolnik B, Spalding MD, Stuart SN, Symes A, Taylor J, Visconti P, Watson JEM, Wood L, Burgess ND (2015) Shortfalls and solutions for meeting national and global conservation area targets. *Conservation Letters*, 8: 329–337.

Brook BW, Traill LW, Bradshaw CJA (2006) Minimum viable population sizes and global extinction risk are unrelated. *Ecology Letters*, 9:375–382.

Clements GR, Bradshaw CJ, Brook BW, Laurance WF (2011) The SAFE index: using a threshold population target to measure relative species threat. *Frontiers in Ecology and the Environment*, 9:521–525.

Di Marco M, Santini L, Visconti P, Mortelliti A, Boitani L, Rondinini C (2016) Using habitat suitability models to scale up population persistence targets for global species conservation. *Hystrix, the Italian Journal of Mammalogy*, 27.

Hilbers JP, Santini L, Visconti P, Schipper AM, Pinto C, Rondinini C, Huijbregts MAJ (2016) Setting population targets for mammals using body mass as a predictor of population persistence. *Conservation Biology*, 31:385–393.

Jamieson IG, Allendorf FW (2012) How does the 50/500 rule apply to MVPs? *Trends in Ecology and Evolution*, 27:578–584.

IUCN (2025) The IUCN Red List of Threatened Species. Version 2025-1. Available at https://www.iucnredlist.org. Accessed on 23 July 2025.

Santini L, Mendez Angarita VY, Karoulis C, Fundarò D, Pranzini N, Vivaldi C, Zhang T, Zampetti A, Gargano SJ, Mirante D, Paltrinieri L (2024) TetraDENSITY 2.0—A database of population density estimates in tetrapods. *Global Ecology and Biogeography*, 33:e13929.

Santini L, Benítez-López A, Dormann CF, Huijbregts MAJ (2022) Population density estimates for terrestrial mammal species. *Global Ecology and Biogeography*, 31:978–994.

Santini L, Tobias JA, Callaghan C, Gallego-Zamorano J, Benítez-López A (2023) Global patterns and predictors of avian population density. *Global Ecology and Biogeography*, 32:1189—1204.

Traill LW, Brook BW, Frankham RR, Bradshaw CJA (2010) Pragmatic population viability targets in a rapidly changing world. *Biological Conservation*, 143:28–34

Traill LW, Bradshaw CJA, Brook BW (2007) Minimum viable population size: A meta-analysis of 30 years of published estimates. *Biological Conservation*, 139:159–166.

Witting L (2024) Population dynamic life history models of the birds and mammals of the world. *Ecological Informatics*, 80:102492.

Wolff NH, Visconti P, Kujala H, Santini L, Hilbers JP, Possingham HP, Oakleaf JR, Kennedy CM, Kiesecker J, Fargione J, Game ET (2023) Prioritizing global land protection for population persistence can double the efficiency of habitat protection for reducing mammal extinction risk. *One Earth*, 6:1564–1575.

See Also

Other target setting methods: spec_absolute_targets(), spec_area_targets(), spec_duran_targets(), spec_interp_absolute_targets(), spec_interp_area_targets(), spec_jung_targets(), spec_max_targets(), spec_min_targets(), spec_polak_targets(), spec_relative_targets(), spec_rl_ecosystem_targets(), spec_rl_species_targets(), spec_rodrigues_targets(), spec_rule_targets(), spec_ward_targets(), spec_watson_targets(), spec_wilson_targets()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_complex_pu_raster <- get_sim_complex_pu_raster()
sim_complex_features <- get_sim_complex_features()

# simulate population density data for each feature,
# expressed as number of individuals per km^2
sim_pop_density_per_km2 <- runif(terra::nlyr(sim_complex_features), 10, 1000)

# create base problem
p0 <-
  problem(sim_complex_pu_raster, sim_complex_features) %>%
  add_min_set_objective() %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# create problem with targets to ensure that, at least, 2500 individuals
# for each feature are represented by the solution
p1 <-
  p0 %>%
  add_auto_targets(
    method = spec_pop_size_targets(
      pop_size = 2500,
      pop_density = sim_pop_density_per_km2,
      density_units = "km^2"
    )
  )

# solve problem
s1 <- solve(p1)

# plot solution
plot(s1, main = "solution based on 2500 population targets", axes = FALSE)
```

```
# create problem with targets to ensure that a particular number of
# individuals for each feature are represented by the solution

# simulate the number of number of individuals required for each feature
target_pop_sizes <- round(
  runif(terra::nlyr(sim_complex_features), 1000, 5000)
 )

# now, create problem with these targets
p2 <-
  p0 %>%
  add_auto_targets(
    method = spec_pop_size_targets(
      pop_size = target_pop_sizes,
      pop_density = sim_pop_density_per_km2,
      density_units = "km^2"
    )
  )

# solve problem
s2 <- solve(p2)

# plot solution
plot(s2, main = "solution based on varying targets", axes = FALSE)

## End(Not run)
```

`spec_relative_targets` *Specify relative targets*

Description

Specify targets expressed as a proportion (between 0 and 1) of the maximum level of representation of each feature 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 values) using the `feature_abundances()` function. Note that this function is designed to be used with `add_auto_targets()` and `add_group_targets()`.

Usage

```
spec_relative_targets(targets, ...)
```

Arguments

`targets`	`numeric` vector that specifies targets for each of the features. If a single `numeric` value is specified, then all features are assigned the same proportion-based target. Note that values range between 0 and 1 (corresponding to 0% and 100% respectively).
`...`	not used.

Value

An object (TargetMethod) for specifying targets that can be used with add_auto_targets() and add_group_targets() to add targets to a problem().

Mathematical formulation

This method involves setting target thresholds based on a proportion. To express this mathematically, we will define the following terminology. Let f denote the total abundance of a feature (e.g., geographic range size), and a the relative target for the feature (per `targets`). Given this terminology, the target threshold (t) for the feature is calculated as follows.

$$t = f \times a$$

See Also

To add relative targets directly to a problem(), see add_relative_targets().

Other target setting methods: spec_absolute_targets(), spec_area_targets(), spec_duran_targets(), spec_interp_absolute_targets(), spec_interp_area_targets(), spec_jung_targets(), spec_max_targets(), spec_min_targets(), spec_polak_targets(), spec_pop_size_targets(), spec_rl_ecosystem_targets(), spec_rl_species_targets(), spec_rodrigues_targets(), spec_rule_targets(), spec_ward_targets(), spec_watson_targets(), spec_wilson_targets()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_complex_pu_raster <- get_sim_complex_pu_raster()
sim_complex_features <- get_sim_complex_features()

# create base problem
p0 <-
  problem(sim_complex_pu_raster, sim_complex_features) %>%
  add_min_set_objective() %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# create problem with targets of 10% for each feature
p1 <-
  p0 %>%
  add_auto_targets(method = spec_relative_targets(targets = 0.1))

# solve problem
s1 <- solve(p1)

# plot solution
plot(s1, main = "solution based on 10% targets", axes = FALSE)

# targets can also be specified for each feature separately.
```

```
# to demonstrate this, we will set a target value for each
# feature based on a random percentage between 10% and 80%
target_values <- runif(terra::nlyr(sim_complex_features), 0.1, 0.8)

# create problem with targets defined separately for each feature
p2 <-
  p0 %>%
  add_auto_targets(method = spec_relative_targets(targets = target_values))

# solve problem
s2 <- solve(p2)

# plot solution
plot(s2, main = "solution based on varying targets", axes = FALSE)

## End(Not run)
```

`spec_rl_ecosystem_targets` *Specify targets based on the IUCN Red List of Ecosystems*

Description

Specify targets based on criteria from the International Union for the Conservation of Nature (IUCN) Red List of Ecosystems (IUCN 2024). Briefly, this method can be used to set targets based on criteria pertaining to geographic distribution size (criterion B) and reductions in geographic distribution size (criterion A). To help prevent widespread features from obscuring priorities for rare features, targets are capped following Butchart *et al.* (2015). This method may be suitable for ecosystem protection at global and and national scales. Note that this function is designed to be used with `add_auto_targets()` and `add_group_targets()`.

Usage

```
spec_rl_ecosystem_targets(
  status,
  criterion_a,
  criterion_b,
  prop_uplift = 0,
  method = "max",
  cap_area_target = 1e+06,
  area_units = "km^2"
)
```

Arguments

`status`	`character` value denoting the IUCN Red List threat status used for target setting. Available options include `"CR"` (Critically Endangered), `"EN"` (Endangered), and `"VU"` (Vulnerable).

`criterion_a` `character` value indicating which subcriterion should be considered based on geographic distribution reduction. Available options include subcriterion based on `"A1"` (reductions over the past 50 years), `"A2a"` (reductions over the next 50 years), `"A2b"` (reductions over any 50 year period), or `"A3"` (reductions during historical time periods). For convenience, these option can also be specified with lower case letters. See Mathematical formulation below for details.

`criterion_b` `character` value indicating which subcriterion should be considered based on geographic distribution size. Available options include subcriterion based on `"B1"` (extent of occupancy), and `"B2"` (area of occupancy). For convenience, these options can also be specified with lower case letters. See Mathematical formulation below for details.

`prop_uplift` `numeric` value denoting the percentage uplift as a proportion. Defaults to 0 (i.e., 0%).

`method` `character` indicating how the target thresholds should be calculated based on values derived from Criterion A and and Criterion B. Available options include `"min"` and `"max"`. For example, `method = "max"` means that the target threshold should be calculated as a maximum of the values from Criterion A and Criterion B. Defaults to `"max"` as a precaution.

`cap_area_target` `numeric` value denoting the area-based target cap. To avoid setting a target cap, a missing (`NA`) value can be specified. Defaults to 1000000 (i.e., 1,000,000 km^2).

`area_units` `character` value denoting the unit of measurement for the area-based arguments. Defaults to `"km^2"` (i.e., km^2).

Details

Targets based on criteria from the IUCN Red List of Ecosystems may be appropriate for global and national scale prioritizations. Despite this, prioritizations based on these criteria may fail to identify meaningful priorities for prioritizations conducted at smaller geographic scales (e.g, local or county scales). For example, if this method is applied to smaller geographic scales, then the resulting prioritizations may select an overly large percentage of the study area, or be biased towards over-representing common and widespread ecosystems. This is because the target thresholds were developed based on criteria for promoting the long-term persistence of entire ecosystems. As such, if you are working at smaller scales, it is recommended to set thresholds based on that criteria are appropriate to the spatial extent of the planning region. Please note that this function is provided as convenient method to set targets for problems with a single management zone, and cannot be used for those with multiple management zones.

Value

An object (`TargetMethod`) for specifying targets that can be used with `add_auto_targets()` and `add_group_targets()` to add targets to a `problem()`.

Mathematical formulation

This method involves setting target thresholds based on assessment criteria from the International Union for the Conservation of Nature (IUCN) Red List of Ecosystems (IUCN 2024). To express

this mathematically, we will define the following terminology. Let f denote the total abundance of a feature (e.g., geographic range size), a the threshold value from Criterion A based on the specified threat status (per `status`, see below for details), b the threshold value from Criterion B based on the specified threat status (per `status`, see below for details), p the percentage uplift as a proportion (per `prop_uplift`), c the target cap (per `cap_area_target` and `area_units`), and $m()$ denote either $max()$ or $min()$ (per `method`). Given this terminology, the target threshold (t) for the feature is calculated as follows.

$$t = min(m(b \times (1 + p), f \times ((1 + p) \times (1 - a))), c, f)$$

Here a and b are equal to one of the following values depending on `status`, `criterion_a`, and `criterion_b`. Note that if `criterion_a` has a value of `"A2a"` or `"A2b"`, then a is assigned the same value as if it were `"A1"`.

- If `status` = `"CR"` and `criterion_a` = `"A1"`, then $a = 80\%$.
- If `status` = `"EN"` and `criterion_a` = `"A1"`, then $a = 50\%$.
- If `status` = `"VU"` and `criterion_a` = `"A1"`, then $a = 30\%$.
- If `status` = `"CR"` and `criterion_a` = `"A3"`, then $a = 90\%$.
- If `status` = `"EN"` and `criterion_a` = `"A3"`, then $a = 70\%$.
- If `status` = `"VU"` and `criterion_a` = `"A3"`, then $a = 30\%$.
- If `status` = `"CR"` and `criterion_b` = `"B1"`, then $b = 2{,}000\ km^2$.
- If `status` = `"EN"` and `criterion_b` = `"B1"`, then $b = 20{,}000\ km^2$.
- If `status` = `"VU"` and `criterion_b` = `"B1"`, then $b = 50{,}000\ km^2$.
- If `status` = `"CR"` and `criterion_b` = `"B2"`, then $b = 200\ km^2$.
- If `status` = `"EN"` and `criterion_b` = `"B2"`, then $b = 2{,}000\ km^2$.
- If `status` = `"VU"` and `criterion_b` = `"B2"`, then $b = 5{,}000\ km^2$.

Data calculations

This function involves calculating targets based on the spatial extent of the features in `x`. Although it can be readily applied to `problem()` objects that have the feature data provided as a `terra::rast()` object, you will need to specify the spatial units for the features when initializing the `problem()` objects if the feature data are provided in a different format. In particular, if the feature data are provided as a `data.frame` or `character` vector, then you will need to specify an argument to `feature_units` when using the `problem()` function. See the Examples section of the documentation for `add_auto_targets()` for a demonstration of specifying the spatial units for features.

References

Butchart SHM, Clarke M, Smith RJ, Sykes RE, Scharlemann JPW, Harfoot M, Buchanan GM, Angulo A, Balmford A, Bertzky B, Brooks TM, Carpenter KE, Comeros-Raynal MT, Cornell J, Ficetola GF, Fishpool LDC, Fuller RA, Geldmann J, Harwell H, Hilton-Taylor C, Hoffmann M, Joolia A, Joppa L, Kingston N, May I, Milam A, Polidoro B, Ralph G, Richman N, Rondinini C, Segan DB, Skolnik B, Spalding MD, Stuart SN, Symes A, Taylor J, Visconti P, Watson JEM, Wood L, Burgess ND (2015) Shortfalls and solutions for meeting national and global conservation area targets. *Conservation Letters*, 8: 329–337.

IUCN (2024) Guidelines for the application of IUCN Red List of Ecosystems Categories and Criteria, Version 2.0. Keith DA, Ferrer-Paris JR, Ghoraba SMM, Henriksen S, Monyeki M, Murray NJ, Nicholson E, Rowland J, Skowno A, Slingsby JA, Storeng AB, Valderrábano M, Zager I (Eds.). Gland, Switzerland: IUCN.

See Also

Other target setting methods: spec_absolute_targets(), spec_area_targets(), spec_duran_targets(), spec_interp_absolute_targets(), spec_interp_area_targets(), spec_jung_targets(), spec_max_targets(), spec_min_targets(), spec_polak_targets(), spec_pop_size_targets(), spec_relative_targets(), spec_rl_species_targets(), spec_rodrigues_targets(), spec_rule_targets(), spec_ward_targets(), spec_watson_targets(), spec_wilson_targets()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data with features that are ecosystem types
tas_pu <- prioritizrdata::get_tas_pu()
tas_features <- prioritizrdata::get_tas_features()

# create base problem
p0 <-
  problem(tas_pu, tas_features, cost_column = "cost") %>%
  add_min_set_objective() %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# note that the following targets will be specified based on subcriterion
# A2 under the assumption that protected areas will be effectively managed,
# and B2 because the feature data (per tas_features) characterize
# area of occupancy

# create problem with targets based on criteria from the IUCN Red List of
# Ecosystems for the Endangered threat status with a 0% uplift
p1 <-
  p0 %>%
  add_auto_targets(
    method = spec_rl_ecosystem_targets(
      status = "EN",
      criterion_a = "A1",
      criterion_b = "B2",
      prop_uplift = 0
    )
  )

# create problem with targets based on criteria from the IUCN Red List of
# Ecosystems for the Endangered threat status with a 20% uplift
p2 <-
  p0 %>%
```

```
  add_auto_targets(
    method = spec_rl_ecosystem_targets(
      status = "EN",
      criterion_a = "A1",
      criterion_b = "B2",
      prop_uplift = 0.2
    )
  )

# create problem with targets based on criteria from the IUCN Red List of
# Ecosystems for the Vulnerable threat status with a 20% uplift
p3 <-
  p0 %>%
  add_auto_targets(
    method = spec_rl_ecosystem_targets(
      status = "VU",
      criterion_a = "A1",
      criterion_b = "B2",
      prop_uplift = 0.2
    )
  )

# solve problems
s <- tas_pu
s$s1 <- solve(p1)$solution_1
s$s2 <- solve(p2)$solution_1
s$s3 <- solve(p3)$solution_1
s <- s[, c("s1", "s2", "s3"), drop = FALSE]

# plot solutions
plot(s, axes = FALSE)

## End(Not run)
```

`spec_rl_species_targets` *Specify targets based on the IUCN Red List of Threatened Species*

Description

Specify targets based on criteria from the IUCN Red List of Threatened Species (IUCN 2025). Briefly, this method can be used to set targets based on criteria pertaining to geographic range size (Criterion B) and population size reduction criteria (Criterion A). To help prevent widespread features from obscuring priorities for rare features, targets are capped following Butchart *et al.* (2015). This method may be suitable for species protection at global and and national scales (e.g., Jung *et al.* 2021, Ward *et al.* 2025). Note that this function is designed to be used with `add_auto_targets()` and `add_group_targets()`.

Usage

```
spec_rl_species_targets(
  status,
  criterion_a,
  criterion_b,
  prop_uplift = 0,
  method = "max",
  cap_area_target = 1e+06,
  area_units = "km^2"
)
```

Arguments

status	character value denoting the IUCN Red List threat status used for target setting. Available options include "CR" (Critically Endangered), "EN" (Endangered), and "VU" (Vulnerable).
criterion_a	character value indicating which subcriterion should be considered based on population size reduction. Available options include subcriterion "A1" (reduction that occurred in the past and the causes are reversible, understood, and have ceased), "A2" (reduction that occurred in the past and the causes may not be reversible, understood, or have ceased); "A3" (reduction is is inferred or suspected in the future), and "A4" (reduction includes a time period in the past and the future and causes may not be reversible, understood, or have ceased). For convenience, these options can also be specified with lower case letters. See Mathematical formulation below for details.
criterion_b	character value indicating which subcriterion should be considered based on geographic range. Available options include subcriterion "B1" (extent of occupancy) and "B2" (area of occupancy). For convenience, these option can also be specified with lower case letters. See Mathematical formulation below for details.
prop_uplift	numeric value denoting the percentage uplift as a proportion. Defaults to 0 (i.e., 0%).
method	character indicating how the target thresholds should be calculated based on values derived from Criterion A and and Criterion B. Available options include "min" and "max". For example, method = "max" means that the target threshold should be calculated as a maximum of the values from Criterion A and Criterion B. Defaults to "max" as a precaution.
cap_area_target	numeric value denoting the area-based target cap. To avoid setting a target cap, a missing (NA) value can be specified. Defaults to 1000000 (i.e., 1,000,000 km^2).
area_units	character value denoting the unit of measurement for the area-based arguments. Defaults to "km^2" (i.e., km^2).

Details

Targets based on criteria from the IUCN Red List of Threatened Species have been applied to global and national scale prioritizations (e.g., Jung *et al.* 2021; Fastré *et al.* 2021; Ward *et al.*

2025). Despite this, prioritizations based on these criteria may fail to identify meaningful priorities for prioritizations conducted at smaller geographic scales (e.g, local or county scales). For example, if this method is applied to smaller geographic scales, then the resulting prioritizations may select an overly large percentage of the study area, or be biased towards over-representing common and widespread species. This is because the target thresholds were developed based on criteria for promoting the long-term persistence of entire species. As such, if you are working at smaller scales, it is recommended to set thresholds based on that criteria are appropriate to the spatial extent of the planning region. Please note that this function is provided as convenient method to set targets for problems with a single management zone, and cannot be used for those with multiple management zones.

Value

An object (`TargetMethod`) for specifying targets that can be used with `add_auto_targets()` and `add_group_targets()` to add targets to a `problem()`.

Mathematical formulation

This method involves setting target thresholds based on assessment criteria from the International Union for the Conservation of Nature (IUCN) Red List of Threatened Species (IUCN 2025). To express this mathematically, we will define the following terminology. Let f denote the total abundance of a feature (e.g., geographic range size), a the threshold value from Criterion A based on the specified threat status (per `status`, see below for details), b the threshold value from Criterion B based on the specified threat status (per `status`, see below for details), p the percentage uplift as a proportion (per `prop_uplift`), c the target cap (per `cap_area_target` and `area_units`), and $m()$ denote either $max()$ or $min()$ (per `method`). Given this terminology, the target threshold (t) for the feature is calculated as follows.

$$t = min(m(b \times (1 + p), f \times ((1 + p) \times (1 - a))), c, f)$$

Here a and b are equal to one of the following values depending on `status`, `criterion_a`, and `criterion_b`. Note that if `criterion_a` has a value of `"A3"` or `"A4"`, then a is assigned the same value as if it were `"A2"`.

- If `status` = `"CR"` and `criterion_a` = `"A1"`, then $a = 90\%$.
- If `status` = `"EN"` and `criterion_a` = `"A1"`, then $a = 70\%$.
- If `status` = `"VU"` and `criterion_a` = `"A1"`, then $a = 50\%$.
- If `status` = `"CR"` and `criterion_a` = `"A2"`, then $a = 80\%$.
- If `status` = `"EN"` and `criterion_a` = `"A2"`, then $a = 50\%$.
- If `status` = `"VU"` and `criterion_a` = `"A2"`, then $a = 30\%$.
- If `status` = `"CR"` and `criterion_b` = `"B1"`, then $b = 100\ km^2$.
- If `status` = `"EN"` and `criterion_b` = `"B1"`, then $b = 5{,}000\ km^2$.
- If `status` = `"VU"` and `criterion_b` = `"B1"`, then $b = 20{,}000\ km^2$.
- If `status` = `"CR"` and `criterion_b` = `"B2"`, then $b = 10\ km^2$.
- If `status` = `"EN"` and `criterion_b` = `"B2"`, then $b = 50\ km^2$.
- If `status` = `"VU"` and `criterion_b` = `"B2"`, then $b = 2{,}000\ km^2$.

Data calculations

This function involves calculating targets based on the spatial extent of the features in `x`. Although it can be readily applied to `problem()` objects that have the feature data provided as a `terra::rast()` object, you will need to specify the spatial units for the features when initializing the `problem()` objects if the feature data are provided in a different format. In particular, if the feature data are provided as a `data.frame` or `character` vector, then you will need to specify an argument to `feature_units` when using the `problem()` function. See the Examples section of the documentation for `add_auto_targets()` for a demonstration of specifying the spatial units for features.

References

Butchart SHM, Clarke M, Smith RJ, Sykes RE, Scharlemann JPW, Harfoot M, Buchanan GM, Angulo A, Balmford A, Bertzky B, Brooks TM, Carpenter KE, Comeros-Raynal MT, Cornell J, Ficetola GF, Fishpool LDC, Fuller RA, Geldmann J, Harwell H, Hilton-Taylor C, Hoffmann M, Joolia A, Joppa L, Kingston N, May I, Milam A, Polidoro B, Ralph G, Richman N, Rondinini C, Segan DB, Skolnik B, Spalding MD, Stuart SN, Symes A, Taylor J, Visconti P, Watson JEM, Wood L, Burgess ND (2015) Shortfalls and solutions for meeting national and global conservation area targets. *Conservation Letters*, 8: 329–337.

Fastré C, van Zeist W-J, Watson JEM, Visconti P (2021) Integrated spatial planning for biodiversity conservation and food production. *One Earth*, 4:1635–1644.

IUCN (2025) The IUCN Red List of Threatened Species. Version 2025-1. Available at https://www.iucnredlist.org. Accessed on 23 July 2025.

Jung M, Arnell A, de Lamo X, García-Rangel S, Lewis M, Mark J, Merow C, Miles L, Ondo I, Pironon S, Ravilious C, Rivers M, Schepaschenko D, Tallowin O, van Soesbergen A, Govaerts R, Boyle BL, Enquist BJ, Feng X, Gallagher R, Maitner B, Meiri S, Mulligan M, Ofer G, Roll U, Hanson JO, Jetz W, Di Marco M, McGowan J, Rinnan DS, Sachs JD, Lesiv M, Adams VM, Andrew SC, Burger JR, Hannah L, Marquet PA, McCarthy JK, Morueta-Holme N, Newman EA, Park DS, Roehrdanz PR, Svenning J-C, Violle C, Wieringa JJ, Wynne G, Fritz S, Strassburg BBN, Obersteiner M, Kapos V, Burgess N, Schmidt- Traub G, Visconti P (2021) Areas of global importance for conserving terrestrial biodiversity, carbon and water. *Nature Ecology and Evolution*, 5:1499–1509.

Mogg S, Fastre C, Jung M, Visconti P (2019) Targeted expansion of Protected Areas to maximise the persistence of terrestrial mammals. *Preprint at bioxriv*, doi:10.1101/608992.

See Also

Other target setting methods: `spec_absolute_targets()`, `spec_area_targets()`, `spec_duran_targets()`, `spec_interp_absolute_targets()`, `spec_interp_area_targets()`, `spec_jung_targets()`, `spec_max_targets()`, `spec_min_targets()`, `spec_polak_targets()`, `spec_pop_size_targets()`, `spec_relative_targets()`, `spec_rl_ecosystem_targets()`, `spec_rodrigues_targets()`, `spec_rule_targets()`, `spec_ward_targets()`, `spec_watson_targets()`, `spec_wilson_targets()`

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
```

```
sim_complex_pu_raster <- get_sim_complex_pu_raster()
sim_complex_features <- get_sim_complex_features()

# create base problem
p0 <-
  problem(sim_complex_pu_raster, sim_complex_features) %>%
  add_min_set_objective() %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# note that the following targets will be specified based on subcriterion
# A2 under the assumption that protected areas will be effectively managed,
# and B2 because the feature data (per sim_complex_features) characterize
# area of occupancy

# create problem with targets based on criteria from the IUCN Red List of
# Threatened Species for the Endangered threat status with a 0% uplift
p1 <-
  p0 %>%
  add_auto_targets(
    method = spec_rl_species_targets(
      status = "EN",
      criterion_a = "A2",
      criterion_b = "B2",
      prop_uplift = 0
    )
  )

# create problem with targets based on criteria from the IUCN Red List of
# Threatened Species for the Endangered threat status with a 20% uplift
p2 <-
  p0 %>%
  add_auto_targets(
    method = spec_rl_species_targets(
      status = "EN",
      criterion_a = "A2",
      criterion_b = "B2",
      prop_uplift = 0.2
    )
  )

# create problem with targets based on criteria from the IUCN Red List of
# Threatened Species for the Vulnerable threat status with a 20% uplift
p3 <-
  p0 %>%
  add_auto_targets(
    method = spec_rl_species_targets(
      status = "VU",
      criterion_a = "A2",
      criterion_b = "B2",
      prop_uplift = 0.2
    )
  )
```

```
# solve problems
s <- c(solve(p1), solve(p2), solve(p3))
names(s) <- c("EN (0%)", "EN (20%)", "VU (20%)")

# plot solutions
plot(s, axes = FALSE)

## End(Not run)
```

`spec_rodrigues_targets` *Specify targets following Rodrigues* et al. *(2004)*

Description

Specify targets based on the methodology outlined by Rodrigues *et al.* (2004). Briefly, this method involves setting targets based on log-linear interpolation methods. To help prevent widespread features from obscuring priorities, targets are capped following Butchart *et al.* (2015). This method was designed for global-scale prioritizations. Note that this function is designed to be used with `add_auto_targets()` and `add_group_targets()`.

Usage

```
spec_rodrigues_targets(
  rare_area_threshold = 1000,
  rare_relative_target = 1,
  common_area_threshold = 250000,
  common_relative_target = 0.1,
  cap_area_target = 1e+06,
  area_units = "km^2"
)
```

Arguments

`rare_area_threshold`
: `numeric` value indicating the threshold area for rare identifying rare features. Defaults to 1000 (i.e., 1000 km^2).

`rare_relative_target`
: `numeric` value denoting the relative target for features with a spatial distribution that is smaller than `rare_area_threshold`. Note that this value must be a proportion between 0 and 1. Defaults to 1 (i.e., 100%).

`common_area_threshold`
: `numeric` value indicating the threshold area for identifying common features. Defaults to 250000 (i.e., 250,000 km^2).

`common_relative_target`
: `numeric` value denoting the relative target for features with a spatial distribution that is greater than `common_area_threshold`. Defaults to 0.1 (i.e., 10%). Since this default value is based on historical levels of global protected area coverage, it may be appropriate to set this value based on current levels of protected area coverage (e.g., 17.6% for terrestrial and 8.4% for marine systems globally; UNEP-WCMC and IUCN 2025).

`cap_area_target`
: `numeric` value denoting the area-based target cap. To avoid setting a target cap, a missing (`NA`) value can be specified. Defaults to 1000000 (i.e., 1,000,000 km^2).

`area_units`
: `character` value denoting the unit of measurement for the area-based arguments. Defaults to `"km^2"` (i.e., km^2).

Details

This target setting method was designed to protect species in global-scale prioritizations (Rodrigues *et al.* 2004). Although it has also been successfully applied to global-scales (e.g., Butchart *et al.* 2015; Hanson *et al.* 2020; Venter *et al.* 2014), it may fail to identify meaningful priorities for prioritizations conducted at smaller geographic scales (e.g., national, state-level or county scales). For example, if this method is applied to such geographic scales, then the resulting prioritizations may select an overly large percentage of the study area, or be biased towards over-representing common and widespread species. This is because the thresholds (i.e., `rare_area_threshold`, `common_area_threshold`, and `cap_area_threshold`) were originally developed based on rationale for promoting the long-term persistence of entire species. As such, if you are working at a sub-global scale, it is recommended to set thresholds based on that criteria are appropriate to the spatial extent of the planning region. Please note that this function is provided as convenient method to set targets for problems with a single management zone, and cannot be used for those with multiple management zones.

Value

An object (`TargetMethod`) for specifying targets that can be used with `add_auto_targets()` and `add_group_targets()` to add targets to a `problem()`.

Mathematical formulation

This method involves setting target thresholds based on the spatial extent of the features. By default, this method identifies rare features as those with a spatial distribution smaller than 1,000 km^2 (per `rare_area_threshold` and `area_units`) and common features as those with a spatial distribution larger than 250,000 km^2 (per `common_area_threshold` and `area_units`). Given this, rare features are assigned a target threshold of 100% (per `rare_relative_target`), common features are assigned a target threshold of 10% (per `common_relative_target`), and features with a spatial distribution that is between the area-based thresholds used to identify rare and common features are assigned a target threshold through log-linear interpolation. Additionally, following Butchart *et al.* (2015), a cap of 1,000,000 km^2 is applied to target thresholds (per `cap_area_threshold` and `area_units`).

Data calculations

This function involves calculating targets based on the spatial extent of the features in `x`. Although it can be readily applied to `problem()` objects that have the feature data provided as a `terra::rast()` object, you will need to specify the spatial units for the features when initializing the `problem()` objects if the feature data are provided in a different format. In particular, if the feature data are provided as a `data.frame` or `character` vector, then you will need to specify an argument to `feature_units` when using the `problem()` function. See the Examples section of the documentation for `add_auto_targets()` for a demonstration of specifying the spatial units for features.

References

Butchart SHM, Clarke M, Smith RJ, Sykes RE, Scharlemann JPW, Harfoot M, Buchanan GM, Angulo A, Balmford A, Bertzky B, Brooks TM, Carpenter KE, Comeros-Raynal MT, Cornell J, Ficetola GF, Fishpool LDC, Fuller RA, Geldmann J, Harwell H, Hilton-Taylor C, Hoffmann M, Joolia A, Joppa L, Kingston N, May I, Milam A, Polidoro B, Ralph G, Richman N, Rondinini C, Segan DB, Skolnik B, Spalding MD, Stuart SN, Symes A, Taylor J, Visconti P, Watson JEM, Wood L, Burgess ND (2015) Shortfalls and solutions for meeting national and global conservation area targets. *Conservation Letters*, 8: 329–337.

Hanson JO, Rhodes JR, Butchart SHM, Buchanan GM, Rondinini C, Ficetola GF, Fuller RA (2020) Global conservation of species' niches. Global conservation of species' niches. *Nature*, 580: 232–234

Rodrigues ASL, Akçakaya HR, Andelman SJ, Bakarr MI, Boitani L, Brooks TM, Chanson JS, Fishpool LDC, Da Fonseca GAB, Gaston KJ, Hoffmann M, Marquet PA, Pilgrim JD, Pressey RL, Schipper J, Sechrest W, Stuart SN, Underhill LG, Waller RW, Watts MEJ, Yan X (2004) Global gap analysis: priority regions for expanding the global protected-area network. *BioScience*, 54: 1092–1100.

UNEP-WCMC and IUCN (2025) Protected Planet Report 2024. Cambridge, UK: UNEP-WCMC and IUCN. Available at <www.protectedplanet.net>.

Venter O, Fuller RA, Segan DB, Carwardine J, Brooks T, Butchart SHM, Di Marco M, Iwamura T, Joseph L, O'Grady D, Possingham HP, Rondinini C, Smith RJ, Venter M, Watson JEM (2014) Targeting global protected area expansion for imperiled biodiversity. *PLoS Biology*, 12: e1001891.

See Also

Other target setting methods: `spec_absolute_targets()`, `spec_area_targets()`, `spec_duran_targets()`, `spec_interp_absolute_targets()`, `spec_interp_area_targets()`, `spec_jung_targets()`, `spec_max_targets()`, `spec_min_targets()`, `spec_polak_targets()`, `spec_pop_size_targets()`, `spec_relative_targets()`, `spec_rl_ecosystem_targets()`, `spec_rl_species_targets()`, `spec_rule_targets()`, `spec_ward_targets()`, `spec_watson_targets()`, `spec_wilson_targets()`

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_complex_pu_raster <- get_sim_complex_pu_raster()
```

```
sim_complex_features <- get_sim_complex_features()

# create problem with Rodrigues et al. (2004) targets
p1 <-
  problem(sim_complex_pu_raster, sim_complex_features) %>%
  add_min_set_objective() %>%
  add_auto_targets(method = spec_rodrigues_targets()) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s1 <- solve(p1)

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

## End(Not run)
```

`spec_rule_targets` *Specify targets following a set of rules*

Description

Specify targets based on a set of rules for ecological and ecosystem criteria. This is a customizable version of the approach in Harris and Holness (2023). To help prevent widespread features from obscuring priorities, targets are capped following Butchart *et al.* (2015). This method was designed to help set targets for a broad range of features (e.g., species, ecosystems, ecosystem services, ecological processes) at local and national scales. Note that this function is designed to be used with `add_auto_targets()` and `add_group_targets()`.

Usage

```
spec_rule_targets(
  baseline_relative_target,
  rules_relative_target,
  data,
  cap_area_target = 1e+06,
  area_units = "km^2"
)
```

Arguments

`baseline_relative_target`

`numeric` value indicating the baseline target values as a proportion (ranging between 0 and 1). Depending on context of the prioritization exercise, a value of 0.3 (i.e., 30%) may be appropriate (Harris *et al.* 2023).

rules_relative_target
: named numeric vector with name-value key pairs denoting the values that should be added together when calculating the relative target for a given feature. Note that values must range between -1 and 1, and the names must correspond to column names of data.

data
: data.frame indicating if each feature meets the criteria for applying each rule (or not) during the target calculations. Here, each row must correspond to a different feature, and each column contains information about the feature (per feature_names(x)). In particular, data must have a "feature" column that contains character values with the feature names. Additionally, data must also contain columns with logical (TRUE/FALSE) values that indicate if the feature meets a particular criterion for calculating targets. For example, data could contain columns that indicate if each feature corresponds to a threatened species, ecosystem service, or culturally important site. Note that if different rules should be applied to species based on their particular threat status (e.g., Critically Endangered and Vulnerable species should receive different target thresholds), then data should contain a column for each threat status (separately).

cap_area_target
: numeric value denoting the area-based target cap. To avoid setting a target cap, a missing (NA) value can be specified. Defaults to 1000000 (i.e., 1,000,000 km^2).

area_units
: character value denoting the unit of measurement for the area-based arguments. Defaults to "km^2" (i.e., km^2).

Details

This method has been applied to set target thresholds at national scales (e.g., Harris *et al.* 2023). It may also be especially useful for local scales. It is based on the rationale that it is appropriate to set the target for a feature based on a linear combination of values. When using this method in a planning exercise, it is important to ensure that the criteria and values used to parameterize the rules reflect the stakeholder objectives. Additionally, the baseline relative target (per baseline_relative_target) and cap threshold (per cap_area_target and area_units) may need to set based on the features and spatial extent of the planning region. Please note that this function is provided as convenient method to set targets for problems with a single management zone, and cannot be used for those with multiple management zones.

Value

An object (TargetMethod) for specifying targets that can be used with add_auto_targets() and add_group_targets() to add targets to a problem().

Mathematical formulation

This method provides a flexible approach for setting target thresholds based on a set of rules. To express this mathematically, we will define the following terminology. Let f denote the total abundance of a feature (e.g., geographic range), b denote the baseline relative target threshold (per baseline_relative_target), and let c denote the cap threshold (per cap_area_target and area_units). To describe the rules, let U denote the set of rules (indexed by u), d_u indicate if the target for the feature should be calculated based on each rule $u \in U$ (using binary values, per

data), and a_u denote the value that should be added to the target given each rule $u \in U$ (per rules_relative_target). Given this terminology, the target threshold (t) for the feature is calculated as follows.

$$t = min(f \times max(min(b + \sum_{u \in U} d_u \times a_u, 1), 0), c)$$

Data calculations

This function involves calculating targets based on the spatial extent of the features in x. Although it can be readily applied to problem() objects that have the feature data provided as a terra::rast() object, you will need to specify the spatial units for the features when initializing the problem() objects if the feature data are provided in a different format. In particular, if the feature data are provided as a data.frame or character vector, then you will need to specify an argument to feature_units when using the problem() function. See the Examples section of the documentation for add_auto_targets() for a demonstration of specifying the spatial units for features.

References

Butchart SHM, Clarke M, Smith RJ, Sykes RE, Scharlemann JPW, Harfoot M, Buchanan GM, Angulo A, Balmford A, Bertzky B, Brooks TM, Carpenter KE, Comeros-Raynal MT, Cornell J, Ficetola GF, Fishpool LDC, Fuller RA, Geldmann J, Harwell H, Hilton-Taylor C, Hoffmann M, Joolia A, Joppa L, Kingston N, May I, Milam A, Polidoro B, Ralph G, Richman N, Rondinini C, Segan DB, Skolnik B, Spalding MD, Stuart SN, Symes A, Taylor J, Visconti P, Watson JEM, Wood L, Burgess ND (2015) Shortfalls and solutions for meeting national and global conservation area targets. *Conservation Letters*, 8: 329–337.

Harris LR, Holness SD (2023) A practical approach to setting heuristic marine biodiversity targets for systematic conservation planning. *Biological Conservation*, 285: 110218.

See Also

Other target setting methods: spec_absolute_targets(), spec_area_targets(), spec_duran_targets(), spec_interp_absolute_targets(), spec_interp_area_targets(), spec_jung_targets(), spec_max_targets(), spec_min_targets(), spec_polak_targets(), spec_pop_size_targets(), spec_relative_targets(), spec_rl_ecosystem_targets(), spec_rl_species_targets(), spec_rodrigues_targets(), spec_ward_targets(), spec_watson_targets(), spec_wilson_targets()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_complex_pu_raster <- get_sim_complex_pu_raster()
sim_complex_features <- get_sim_complex_features()

# calculate total distribution size for features in km^2
feature_size <- as.numeric(units::set_units(
  units::set_units(
    terra::global(sim_complex_features, "sum", na.rm = TRUE)[[1]] *
```

```
      prod(terra::res(sim_complex_features)),
    "m^2"
  ),
  "km^2"
))

# simulate data that provide additional information for each feature
rule_data <- tibble::tibble(feature = names(sim_complex_features))

# add a column indicating if each feature has a small distribution,
rule_data$small_distribution <- feature_size <= 1000

# add a column indicating if each feature has a large distribution,
rule_data$large_distribution <- feature_size >= 5000

# add a column indicating if each feature has low quality data
# associated with it, based on random simulated values
rule_data$low_quality <- sample(
  c(TRUE, FALSE), terra::nlyr(sim_complex_features),
  replace = TRUE, prob = c(0.2, 0.8)
)

# next, we will add simulate dat for columns indicating the threat status of
# the features. since all columns must contain logical (TRUE/FALSE) values,
# we will add a column for each threat status separately. note that these
# values will be simulated such that a feature will only have a value of
# TRUE for, at most, a single threat status

# add a column indicating if each feature has a Vulnerable threat status
rule_data$vulnerable <- sample(
  c(TRUE, FALSE), terra::nlyr(sim_complex_features),
  replace = TRUE, prob = c(0.3, 0.7)
)

# add a column indicating if each feature has an Endangered threat status,
# based on random simulated values
rule_data$endangered <- sample(
  c(TRUE, FALSE), terra::nlyr(sim_complex_features),
  replace = TRUE, prob = c(0.3, 0.7)
) & !rule_data$vulnerable

# add a column indicating if each feature has a Critically Endangered threat
# status, based on random simulated values
rule_data$critically_endangered <- sample(
  c(TRUE, FALSE), terra::nlyr(sim_complex_features),
  replace = TRUE, prob = c(0.3, 0.7)
) & !rule_data$endangered & !rule_data$vulnerable

# preview rule data
print(rule_data)

# create problem with rule based targets, wherein targets are calculated
# with a baseline of 30%, features with a small distribution are assigned
```

```
# targets of an additional 10%, features with a large distribution are
# assigned targets reduced by 10%, features with low quality data are
# assigned targets reduced by 10%, features with a Vulnerable threat status
# are assigned targets of an additional 5%, features with an Endangered
# threat status are assigned targets of an additional 10%, and features with
# a Critically Endangered threat status are assigned targets of an
# additional 20%
p1 <-
  problem(sim_complex_pu_raster, sim_complex_features) %>%
  add_min_set_objective() %>%
  add_auto_targets(
    method = spec_rule_targets(
      baseline_relative_target = 0.3,
      rules_relative_target = c(
        "small_distribution" = 0.1,
        "large_distribution" = -0.1,
        "low_quality" = -0.1,
        "vulnerable" = 0.05,
        "endangered" = 0.1,
        "critically_endangered" = 0.2
      ),
      data = rule_data
    )
  ) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s1 <- solve(p1)

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

## End(Not run)
```

`spec_ward_targets` *Specify targets following Ward* et al. *(2025)*

Description

Specify targets based on the methodology outlined by Ward *et al.* (2025). Briefly, this method involves setting targets based the criteria for recognizing Critically Endangered species by the International Union for the Conservation of Nature (IUCN) Red List of Threatened Species (IUCN 2025). To help prevent widespread features from obscuring priorities, targets are capped following Butchart *et al.* (2015). This method was designed for species protection at national-scales. Note that this function is designed to be used with `add_auto_targets()` and `add_group_targets()`.

Usage

```
spec_ward_targets(status = "CR", cap_area_target = 1e+06, area_units = "km^2")
```

Arguments

`status`	`character` value denoting the IUCN Red List threat status used for target setting. Available options include `"CR"` (Critically Endangered) , `"EN"` (Endangered), and `"VU"` (Vulnerable). Defaults to `"CR"`.
`cap_area_target`	`numeric` value denoting the area-based target cap. To avoid setting a target cap, a missing (`NA`) value can be specified. Defaults to 1000000 (i.e., 1,000,000 km^2).
`area_units`	`character` value denoting the unit of measurement for the area-based arguments. Defaults to `"km^2"` (i.e., km^2).

Details

This target setting method was designed to protect species in a national-scale prioritizations (Ward *et al.* 2025). Since it was designed for national-scale prioritizations, it may fail to identify meaningful priorities for prioritizations conducted at smaller geographic scales (e.g., national, state-level or county scales). For example, if this method is applied to smaller geographic scales, then the resulting prioritizations may select an overly large percentage of the study area, or be biased towards over-representing common and widespread species. As such, if you are working at smaller scales, it is recommended to set thresholds based on that criteria are appropriate to the spatial extent of the planning region. Please note that this function is provided as convenient method to set targets for problems with a single management zone, and cannot be used for those with multiple management zones.

Value

An object (`TargetMethod`) for specifying targets that can be used with `add_auto_targets()` and `add_group_targets()` to add targets to a `problem()`.

Mathematical formulation

This method involves setting target thresholds based on assessment criteria from the IUCN Red List (IUCN 2025). It is based on the rationale that protected areas prevent the local extinction of populations located inside them, and so a protected area system can safeguard enough of a species' distribution to ensure that the species – in event that it becomes locally extinct outside of protected areas – would, at worst, be classified under a particular threat status. In particular, this method considers criteria related to the size of a species' spatial distribution (i.e., Criterion B) and population size reduction (i.e., Criterion A). By default, it considers criteria for the Critically Endangered threat status and involves setting the target threshold for a species as 100,000 km^2 (per subcriterion B1) or 20% (per subcriterion A2) of its spatial distribution (which ever value is larger). Additionally, following Butchart *et al.* (2015), a cap of 1,000,000 km^2 is applied to target thresholds (per `cap_area_threshold` and `area_units`). By helping to ensure that species would – at a minimum – meet criteria for being recognized as Critically Endangered, this method aims to reduce chance that species will become extinct.

Data calculations

This function involves calculating targets based on the spatial extent of the features in `x`. Although it can be readily applied to `problem()` objects that have the feature data provided as a `terra::rast()`

object, you will need to specify the spatial units for the features when initializing the problem() objects if the feature data are provided in a different format. In particular, if the feature data are provided as a data.frame or character vector, then you will need to specify an argument to feature_units when using the problem() function. See the Examples section of the documentation for add_auto_targets() for a demonstration of specifying the spatial units for features.

References

Butchart SHM, Clarke M, Smith RJ, Sykes RE, Scharlemann JPW, Harfoot M, Buchanan GM, Angulo A, Balmford A, Bertzky B, Brooks TM, Carpenter KE, Comeros-Raynal MT, Cornell J, Ficetola GF, Fishpool LDC, Fuller RA, Geldmann J, Harwell H, Hilton-Taylor C, Hoffmann M, Joolia A, Joppa L, Kingston N, May I, Milam A, Polidoro B, Ralph G, Richman N, Rondinini C, Segan DB, Skolnik B, Spalding MD, Stuart SN, Symes A, Taylor J, Visconti P, Watson JEM, Wood L, Burgess ND (2015) Shortfalls and solutions for meeting national and global conservation area targets. *Conservation Letters*, 8: 329–337.

IUCN (2025) The IUCN Red List of Threatened Species. Version 2025-1. Available at https://www.iucnredlist.org. Accessed on 23 July 2025.

Ward M, Possingham HP, Wintle BA, Woinarski JCZ, Marsh JR, Chapple DG, Lintermans M, Scheele BC, Whiterod NS, Hoskin CJ, Aska B, Yong C, Tulloch A, Stewart R, Watson JEM (2025) The estimated cost of preventing extinction and progressing recovery for Australia's priority threatened species. *Proceedings of the National Academy of Sciences*, 122: e2414985122.

See Also

Other target setting methods: spec_absolute_targets(), spec_area_targets(), spec_duran_targets(), spec_interp_absolute_targets(), spec_interp_area_targets(), spec_jung_targets(), spec_max_targets(), spec_min_targets(), spec_polak_targets(), spec_pop_size_targets(), spec_relative_targets(), spec_rl_ecosystem_targets(), spec_rl_species_targets(), spec_rodrigues_targets(), spec_rule_targets(), spec_watson_targets(), spec_wilson_targets()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_complex_pu_raster <- get_sim_complex_pu_raster()
sim_complex_features <- get_sim_complex_features()

# create problem with Ward et al. (2025) targets
p1 <-
  problem(sim_complex_pu_raster, sim_complex_features) %>%
  add_min_set_objective() %>%
  add_auto_targets(method = spec_ward_targets()) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

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

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

## End(Not run)
```

`spec_watson_targets` *Specify targets following Watson* et al. *(2010)*

Description

Specify targets based on the methodology outlined by Watson *et al.* (2010). Briefly, this method involves setting targets thresholds as a percentage based on whether or not a feature is considered rare. To help prevent widespread features from obscuring priorities, targets are capped following Butchart *et al.* (2015). This method was designed for species protection at national-scales. Note that this function is designed to be used with `add_auto_targets()` and `add_group_targets()`.

Usage

```
spec_watson_targets(
  rare_area_threshold = 10000,
  rare_relative_target = 1,
  rare_area_target = 1000,
  common_relative_target = 0.1,
  cap_area_target = 1e+06,
  area_units = "km^2"
)
```

Arguments

`rare_area_threshold`
: `numeric` value indicating the threshold area for rare identifying rare features. Defaults to 10000 (i.e., 10000 km^2).

`rare_relative_target`
: `numeric` value denoting the relative target for features with a spatial distribution that is smaller than `rare_area_threshold`. Note that this value must be a proportion between 0 and 1. Defaults to 1 (i.e., 100%).

`rare_area_target`
: `numeric` value indicating the area-based target for features with a spatial distribution that is smaller than `rare_area_threshold`. Defaults to 1000 (i.e., 1000 km^2).

`common_relative_target`
: `numeric` value denoting the relative target for features with a spatial distribution that is greater than `common_area_threshold`. Defaults to 0.1 (i.e., 10%). Since this default value is based on historical levels of global protected area coverage, it may be appropriate to set this value based on current levels of protected area coverage (e.g., 17.6% for terrestrial and 8.4% for marine systems globally; UNEP-WCMC and IUCN 2025).

`cap_area_target`
: `numeric` value denoting the area-based target cap. To avoid setting a target cap, a missing (`NA`) value can be specified. Defaults to 1000000 (i.e., 1,000,000 km^2).

`area_units`
: `character` value denoting the unit of measurement for the area-based arguments. Defaults to `"km^2"` (i.e., km^2).

Details

This target setting method was designed to protect species in a national-scale prioritization (Watson *et al.* 2010). Although similar methods have also been successfully to national-scale prioritizations (e.g., Kark *et al.* 2009), it may fail to identify meaningful priorities for prioritizations conducted at smaller geographic scales (e.g., local or county-level scales). For example, if this target setting method is applied to such geographic scales, then the resulting prioritizations may select an overly large percentage of the study area, or be biased towards over-representing common and widespread species. This is because the thresholds (i.e., `rare_area_threshold` and `cap_area_threshold`) were originally developed based on criteria for national-scales. As such, if you working at the sub-national scale, it is recommended to set thresholds based on that criteria are appropriate to the spatial extent of the planning region. Please note that this function is provided as convenient method to set targets for problems with a single management zone, and cannot be used for those with multiple management zones.

Value

An object (`TargetMethod`) for specifying targets that can be used with `add_auto_targets()` and `add_group_targets()` to add targets to a `problem()`.

Mathematical formulation

This method involves setting target thresholds based on the spatial extent of the features. By default, this method identifies rare features as those with a spatial distribution smaller than 10,000 km^2 (per `rare_area_threshold` and `area_units`) and common features as those with a larger spatial distribution. Given this, rare features are assigned target threshold of 100% (per `rare_relative_target`) or 1,000 km^2 (per `rare_area_threshold`) (whichever of these two values is smaller), and common features are assigned a target threshold of 10% (per `common_relative_target`). Additionally, following Butchart *et al.* (2015), a cap of 1,000,000 km^2 is applied to target thresholds (per `cap_area_threshold` and `area_units`).

Data calculations

This function involves calculating targets based on the spatial extent of the features in `x`. Although it can be readily applied to `problem()` objects that have the feature data provided as a `terra::rast()` object, you will need to specify the spatial units for the features when initializing the `problem()` objects if the feature data are provided in a different format. In particular, if the feature data are provided as a `data.frame` or `character` vector, then you will need to specify an argument to `feature_units` when using the `problem()` function. See the Examples section of the documentation for `add_auto_targets()` for a demonstration of specifying the spatial units for features.

References

Butchart SHM, Clarke M, Smith RJ, Sykes RE, Scharlemann JPW, Harfoot M, Buchanan GM, Angulo A, Balmford A, Bertzky B, Brooks TM, Carpenter KE, Comeros-Raynal MT, Cornell J, Ficetola GF, Fishpool LDC, Fuller RA, Geldmann J, Harwell H, Hilton-Taylor C, Hoffmann M, Joolia A, Joppa L, Kingston N, May I, Milam A, Polidoro B, Ralph G, Richman N, Rondinini C, Segan DB, Skolnik B, Spalding MD, Stuart SN, Symes A, Taylor J, Visconti P, Watson JEM, Wood L, Burgess ND (2015) Shortfalls and solutions for meeting national and global conservation area targets. *Conservation Letters*, 8: 329–337.

Kark S, Levin N, Grantham HS, Possingham HP (2009) Between-country collaboration and consideration of costs increase conservation planning efficiency in the Mediterranean Basin. *Proceedings of the National Academy of Sciences*, 106: 15368–15373.

UNEP-WCMC and IUCN (2025) Protected Planet Report 2024. Cambridge, UK: UNEP-WCMC and IUCN. Available at <www.protectedplanet.net>.

Watson JEM, Evans MC, Carwardine J, Fuller RA, Joseph LN, Segan DB, Taylor MFJ, Fensham RJ, Possingham HP (2010) The capacity of Australia's protected-area system to represent threatened species. *Conservation Biology*,25: 324–332.

See Also

Other target setting methods: spec_absolute_targets(), spec_area_targets(), spec_duran_targets(), spec_interp_absolute_targets(), spec_interp_area_targets(), spec_jung_targets(), spec_max_targets(), spec_min_targets(), spec_polak_targets(), spec_pop_size_targets(), spec_relative_targets(), spec_rl_ecosystem_targets(), spec_rl_species_targets(), spec_rodrigues_targets(), spec_rule_targets(), spec_ward_targets(), spec_wilson_targets()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_complex_pu_raster <- get_sim_complex_pu_raster()
sim_complex_features <- get_sim_complex_features()

# create problem with Watson et al. (2010) targets
p1 <-
  problem(sim_complex_pu_raster, sim_complex_features) %>%
  add_min_set_objective() %>%
  add_auto_targets(method = spec_watson_targets()) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)

# solve problem
s1 <- solve(p1)

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

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

`spec_wilson_targets` *Specify targets following Wilson* et al. *(2010)*

Description

Specify targets based on the methodology outlined by Wilson *et al.* (2010). Briefly, this method involves using population growth rate data to set target thresholds based on the amount of habitat required to sustain populations for 100,000 years. To help prevent widespread features from obscuring priorities, targets are capped following Butchart *et al.* (2015). Note that this function is designed to be used with `add_auto_targets()` and `add_group_targets()`.

Usage

```
spec_wilson_targets(
  mean_growth_rates,
  var_growth_rates,
  pop_density,
  density_units,
  cap_area_target = 1e+06,
  area_units = "km^2"
)
```

Arguments

`mean_growth_rates`
: `numeric` vector that specifies the average population growth rate that would be expected for each feature within priority areas (i.e., r per Wilson *et al.* 2010). If a single `numeric` value is specified, then all features are assigned targets based on the same average population growth rate. See Population growth section for more details.

`var_growth_rates`
: `numeric` vector that specifies the variance in population growth rate that would be expected for each feature within priority areas (i.e., σ^2 per Wilson *et al.* 2010). If a single `numeric` value is specified, then all features are assigned targets assuming the same variance in population growth rate. See Population growth section for more details.

`pop_density`
: `numeric` vector that specifies the population density for each feature. If a single `numeric` value is specified, then all features are assigned targets assuming the same population density. See Population density section for more details.

`density_units`
: `character` vector that specifies the area-based units for the population density values. For example, units can be used to express that population densities are in terms of individuals per hectare (`"ha"`), acre (`"acre"`), or km^2 (`"km^2"`). If a single `character` value is specified, then all features are assigned targets assuming that population density values are in the same units. See Population density section for more details.

`cap_area_target`
: `numeric` value denoting the area-based target cap. To avoid setting a target cap, a missing (`NA`) value can be specified. Defaults to 1000000 (i.e., 1,000,000 km^2).

`area_units`
: `character` value denoting the unit of measurement for the area-based arguments. Defaults to `"km^2"` (i.e., km^2).

Details

This target setting method was developed to identify the minimum amount of habitat to protect an entire species (Wilson *et al.* 2010). Although this method was originally applied to the sub-national scale (i.e., the East Kalimantan province of Indonesia), Wilson *et al.* (2010) linearly re-scaled targets derived from this method according to the proportion of each species' distribution located within the study area (i.e., based on the species' total distribution size in Borneo). As such, this method may especially well-suited for national or global-scale conservation planning exercises, and may also be useful for local-scale planning exercises as long as the targets are re-scaled appropriately. Please note that this function is provided as convenient method to set targets for problems with a single management zone, and cannot be used for those with multiple management zones.

Value

An object (`TargetMethod`) for specifying targets that can be used with `add_auto_targets()` and `add_group_targets()` to add targets to a `problem()`.

Population growth

This method requires population growth rate data. Although the package does not provide such data, population growth rate estimates can be obtained from published datasets (e.g., Brook *et al.* 2006). Additionally, population growth rate data may be approximated from physiological traits (e.g., such as body mass, Sinclair 1996; Hilbers *et al.* 2016). Indeed, Wilson *et al.* (2010) detail equations for approximating average population growth rate and variance in population growth rate for mammal species based on body mass (based on Sinclair 1996).

Mathematical formulation

This method involves setting target thresholds based on the amount of habitat required to sustain populations for 100,000 years. To express this mathematically, we will define the following terminology. Let f denote the total abundance of a feature (i.e., geographic range size expressed as km^2), k the carrying capacity required for a population to persist for 100,000 years, d the population density of the feature (i.e., number of individuals per km^2, per `pop_density` and `density_units`), r the mean population growth rate of the feature inside protected areas (per `mean_growth_rates`), σ^2 the variance in population growth rate of the feature inside protected areas (per `var_growth_rates`), b is a constant calculated from r and σ^2, and j the target cap (expressed as km^2, per `cap_area_target` and `area_units`). Given this terminology, the target threshold (t) for the feature is calculated as follows.

$$t = min(f, min(j, d \times k))k = (\frac{100000 \times \sigma^2 \times b^2}{2})^{\frac{1}{b}}b = (2 \times \frac{r}{\sigma^2}) - 1$$

Data calculations

This function involves calculating targets based on the spatial extent of the features in `x`. Although it can be readily applied to `problem()` objects that have the feature data provided as a `terra::rast()` object, you will need to specify the spatial units for the features when initializing the `problem()` objects if the feature data are provided in a different format. In particular, if the feature data are provided as a `data.frame` or `character` vector, then you will need to specify an argument to `feature_units` when using the `problem()` function. See the Examples section of the documentation for `add_auto_targets()` for a demonstration of specifying the spatial units for features.

Population density

This method requires population density data expressed as the number of individuals per unit area. For example, if a species has 200 individuals per hectare, then this can be specified with `pop_density = 200` and `density_units = "ha"`. Alternatively, if a species has a population density where one individual occurs every 10 km^2, then this can be specified with `pop_density = 0.1` and `density_units = "km^2"`. Also, note that population density is assumed to scale linearly with the values in the feature data. For example, if a planning unit contains 5 km^2 of habitat for a feature, `pop_density = 200`, and `density_units = "km^2"`, then the calculations assume that the planning unit contains 100 individuals for the species. Although the package does not provide the population density data required to apply this target setting method, such data can be obtained from published databases (e.g., Santini *et al.* 2022, 2023, 2024; Witting *et al.* 2024).

References

Brook BW, Traill LW, Bradshaw CJA (2006) Minimum viable population sizes and global extinction risk are unrelated. *Ecology Letters*, 9:375–382.

Butchart SHM, Clarke M, Smith RJ, Sykes RE, Scharlemann JPW, Harfoot M, Buchanan GM, Angulo A, Balmford A, Bertzky B, Brooks TM, Carpenter KE, Comeros-Raynal MT, Cornell J, Ficetola GF, Fishpool LDC, Fuller RA, Geldmann J, Harwell H, Hilton-Taylor C, Hoffmann M, Joolia A, Joppa L, Kingston N, May I, Milam A, Polidoro B, Ralph G, Richman N, Rondinini C, Segan DB, Skolnik B, Spalding MD, Stuart SN, Symes A, Taylor J, Visconti P, Watson JEM, Wood L, Burgess ND (2015) Shortfalls and solutions for meeting national and global conservation area targets. *Conservation Letters*, 8: 329–337.

Hilbers JP, Santini L, Visconti P, Schipper AM, Pinto C, Rondinini C, Huijbregts MAJ (2016) Setting population targets for mammals using body mass as a predictor of population persistence. *Conservation Biology*, 31:385–393.

IUCN (2025) The IUCN Red List of Threatened Species. Version 2025-1. Available at https://www.iucnredlist.org. Accessed on 23 July 2025.

Santini L, Mendez Angarita VY, Karoulis C, Fundarò D, Pranzini N, Vivaldi C, Zhang T, Zampetti A, Gargano SJ, Mirante D, Paltrinieri L (2024) TetraDENSITY 2.0—A database of population density estimates in tetrapods. *Global Ecology and Biogeography*, 33:e13929.

Santini L, Benítez-López A, Dormann CF, Huijbregts MAJ (2022) Population density estimates for terrestrial mammal species. *Global Ecology and Biogeography*, 31:978–994.

Santini L, Tobias JA, Callaghan C, Gallego-Zamorano J, Benítez-López A (2023) Global patterns and predictors of avian population density. *Global Ecology and Biogeography*, 32:1189—1204.

Sinclair ARE (1996) Mammal populations: fluctuation, regulation, life history theory and their implications for conservation. In Frontiers of Population Ecology. Floyd RB,Sheppard AW, de Barro PJ (Eds.). Melbourne, Australia: CSIRO Publishing.

Wilson KA, Meijaard E, Drummond S, Grantham HS, Boitani L, Catullo G, Christie L, Dennis R, Dutton I, Falcucci A, Maiorano L, Possingham HP, Rondinini C, Turner WR, Venter O, Watts M (2010) Conserving biodiversity in production landscapes. *Ecological Applications*, 20:1721–1732.

Witting L (2024) Population dynamic life history models of the birds and mammals of the world. *Ecological Informatics*, 80:102492.

See Also

Other target setting methods: spec_absolute_targets(), spec_area_targets(), spec_duran_targets(), spec_interp_absolute_targets(), spec_interp_area_targets(), spec_jung_targets(), spec_max_targets(), spec_min_targets(), spec_polak_targets(), spec_pop_size_targets(), spec_relative_targets(), spec_rl_ecosystem_targets(), spec_rl_species_targets(), spec_rodrigues_targets(), spec_rule_targets(), spec_ward_targets(), spec_watson_targets()

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_complex_pu_raster <- get_sim_complex_pu_raster()
sim_complex_features <- get_sim_complex_features()

# simulate mean population growth rate data for each feature
sim_mean_growth_rates <- runif(terra::nlyr(sim_complex_features), 1, 3.0)

# simulate variance in population growth rate data for each feature
sim_var_growth_rates <- runif(terra::nlyr(sim_complex_features), 1.0, 2.0)

# simulate population density data for each feature,
# expressed as number of individuals per km^2
sim_pop_density_per_km2 <- runif(terra::nlyr(sim_complex_features), 10, 100)

# create problem with targets based on Wilson et al. (2010)
p1 <-
  problem(sim_complex_pu_raster, sim_complex_features) %>%
  add_min_set_objective() %>%
  add_auto_targets(
    method = spec_wilson_targets(
      mean_growth_rate = sim_mean_growth_rates,
      var_growth_rates = sim_var_growth_rates,
      pop_density = sim_pop_density_per_km2,
      density_units = "km^2"
    )
  ) %>%
  add_binary_decisions() %>%
  add_default_solver(verbose = FALSE)
```

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

# plot solution
plot(s1, main = "solution", axes = 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 that 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 using symmetric data.

`eval_asym_connectivity_summary()` Calculate the connectivity held within a solution using asymmetric data.

See Also

Other overviews: `constraints`, `decisions`, `importance`, `objectives`, `penalties`, `portfolios`, `solvers`, `targets`

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_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)

# 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 symmetric 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 using symmetric data
eval_connectivity_summary(p, s, data = cm)

# create an asymmetric connectivity matrix to describe pair-wise
# connectivity values between combinations of planning units

# for brevity, we will just generate a matrix with random values
acm <- matrix(
  runif(ncell(sim_pu_raster) ^ 2),
  ncol = terra::ncell(sim_pu_raster)
)
```

```
# evaluate connectivity of solution using asymmetric data
eval_asym_connectivity_summary(p, s, data = acm)


## End(Not run)
```

`Target-class` *Target class*

Description

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

Super class

`prioritizr::ConservationModifier` -> `Target`

Methods

Public methods:

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

Method `output()`**:** Output the targets.

Usage:

```
Target$output()
```

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

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other classes: `ConservationModifier-class`, `ConservationProblem-class`, `Constraint-class`, `Decision-class`, `Objective-class`, `OptimizationProblem-class`, `Penalty-class`, `Portfolio-class`, `Solver-class`, `TargetMethod-class`, `Weight-class`

`TargetMethod-class` *Target setting method class*

Description

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

Public fields

`name` `character` value with name of method.

`type` `character` value denoting the target type.

`fun` `function` for calculating targets.

`args` `list` containing arguments.

`frame` defused 'call for generating error messages.

Methods

Public methods:

- `TargetMethod$new()`
- `TargetMethod$print()`
- `TargetMethod$calculate_targets()`
- `TargetMethod$calculate_targets_km2()`
- `TargetMethod$calculate_relative_targets()`
- `TargetMethod$calculate_absolute_targets()`
- `TargetMethod$clone()`

Method `new()`**:** Initialize new object.

Usage:

```
TargetMethod$new(name, type, fun, args, frame)
```

Arguments:

`name` `character` value with name of method.

`type` `character` value denoting the target type. Available options include `"relative"` and `"absolute"`.

`fun` `function` for calculating targets.

`args` `list` containing arguments.

`frame` defused 'call for generating error messages.

Returns: A new `Method` object.

Method `print()`**:** Print the object.

Usage:

```
TargetMethod$print(...)
```

Arguments:

... not used.

Returns: Invisible TRUE.

Method `calculate_targets()`**:** Calculate targets expressed in the type of units defined for the method (per $type).

Usage:

```
TargetMethod$calculate_targets(x, features, call = NULL)
```

Arguments:

x problem() object.

features integer feature indices.

call NULL or calling environment.

Returns: A numeric vector with target values.

Method `calculate_targets_km2()`**:** Calculate targets as km^2.

Usage:

```
TargetMethod$calculate_targets_km2(x, features, call = NULL)
```

Arguments:

x problem() object.

features integer feature indices.

call NULL or calling environment.

Returns: A numeric vector with target values expressed in km^2.

Method `calculate_relative_targets()`**:** Calculate targets as km^2.

Usage:

```
TargetMethod$calculate_relative_targets(x, features, call = NULL)
```

Arguments:

x problem() object.

features integer feature indices.

call NULL or calling environment.

Returns: A numeric vector with target values expressed as relative units.

Method `calculate_absolute_targets()`**:** Calculate targets expressed as absolute units.

Usage:

```
TargetMethod$calculate_absolute_targets(x, features, call = NULL)
```

Arguments:

x problem() object.

features integer feature indices.

call NULL or calling environment.

Returns: A numeric vector with target values expressed as absolute units.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

Other classes: ConservationModifier-class, ConservationProblem-class, Constraint-class, Decision-class, Objective-class, OptimizationProblem-class, Penalty-class, Portfolio-class, Solver-class, Target-class, Weight-class

---

`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. Most objectives require targets, and attempting to solve a problem that requires targets and does not have them specified will result in an error.

Details

The following functions can be used to add targets to a conservation planning problem(). For further information on these functions, see the Usage section below. Also, note that if multiple of these functions are added to a problem(), then only the last function added will be used.

add_auto_targets() Add targets based on a particular target setting method.

add_group_targets() Add targets wherein each feature is assigned to a particular group and a target setting method is specified for each feature group.

add_relative_targets() Add targets expressed as a proportion (between 0 and 1) of the maximum level of representation of each feature in the study area.

add_absolute_targets() Add targets expressed as the same values as the underlying feature data (ignoring any specified feature units).

add_manual_targets() Add targets by manually specifying all the required information for each target. This function can be used customize all aspects of a target and is especially useful when working with multiple zones.

Target setting

Many conservation planning problems require targets. Targets are used to specify the minimum amount, or proportion, of a feature's spatial distribution that should ideally be protected. This is important so that the optimization process can weigh the merits and trade-offs between improving the representation of one feature over another feature. Although it can be challenging to set meaningful targets, this is a critical step for ensuring that prioritizations meet the stakeholder objectives that underpin a prioritization exercise (Carwardine *et al.* 2009). In other words, targets play an important role in ensuring that a priority setting process is properly tuned according to stakeholder requirements. For example, targets provide a mechanism for ensuring that a prioritization secures enough habitat to promote the long-term persistence of each threatened species, culturally important species, or economically important ecosystem services under consideration. Since there is often uncertainty regarding stakeholder objectives (e.g., how much habitat should be protected for a given species) or the influence of particular target on a prioritization (e.g., how would setting a 90% or 100% for a threatened species alter priorities), it is often useful to generate and compare a suite of prioritizations based on different target scenarios.

Usage

A variety of functions can be used to specify targets for a conservation planning problem. Below we describe them in detail.

- The add_auto_targets() function can be used to add targets to a conservation planning problem. It provides a flexible interface for specifying targets based on a single method, or a combination of methods. Note that this function is specifically designed for problems that have a single zone, and cannot be used for problems that have multiple zones. For example, given problem x, it could be used to specify targets for all features calculated following Jung *et al.* (2021) with the following code.

```
# specify targets based on default parameters via method name
x %>% add_auto_targets(method = "jung")

# specify targets based on default parameters via function
x %>% add_auto_targets(method = spec_jung_targets())

# specify targets based on customized parameters via function
x %>% add_auto_targets(method = spec_jung_targets(prop_uplift = 0.05))
```

Additionally, if x had three features – with the first feature corresponding to an ecosystem and the latter two to different species – this function could be used to specify a target for the ecosystem feature based on Polak *et al.* (2015) and targets for the species features based on Jung *et al.* (2021) with the following code.

```
# specify target setting methods for each feature with default parameters
# via method name
x %>% add_auto_targets(
  method = list("polak", "jung", "jung")
)

# specify target setting methods for each feature with customized parameters
# via functions
x %>% add_auto_targets(
  method = list(
    spec_polak_targets(),
    spec_jung_targets(prop_uplift = 0.05),
    spec_jung_targets(prop_uplift = 0.07)
  )
)
```

Targets can also be specified based on the maximum or minimum of multiple other target setting methods. For example, targets can be specified as the maximum of the Polak *et al.* (2015) and Jung *et al.* (2021) target setting methodologies with the following code:

```
x %>% add_auto_targets(
  method = spec_max_targets(spec_polak_targets(), spec_jung_targets())
)
```

The following functions can be used to specify targets in conjunction with the add_auto_targets() function. Note that some of these functions do not have default parameters for all arguments

and, as such, cannot be specified using their name (e.g., `method = "relative"` will not work because `spec_relative_targets()` requires the user to specify an argument for the `targets` parameter).

`spec_relative_targets()` Specify targets expressed as a proportion (between 0 and 1) of the total amount of each feature.

`spec_absolute_targets()` Specify targets expressed as the same values as the underlying feature data (ignoring any specified feature units).

`spec_area_targets()` Specify targets expressed as area-based units.

`spec_interp_absolute_targets()` Specify targets based on an interpolation procedure between thresholds expressed as the same values as the underlying feature data (ignoring any specified feature units).

`spec_interp_area_targets()` Specify targets based on an interpolation procedure between area-based thresholds.

`spec_pop_size_targets()` Specify targets based on minimum population sizes.

`spec_rule_targets()` Specify targets calculated following a rule-based procedure based on a set of ecological and ecosystem criteria. This is a customizable version of the approach in Harris and Holness (2023).

`spec_rl_species_targets()` Specify targets based on criteria from the International Union for the Conservation of Nature (IUCN) Red List of Threatened Species (IUCN 2025).

`spec_rl_ecosystem_targets()` Specify targets based on criteria from the International Union for the Conservation of Nature (IUCN) Red List of Ecosystems (IUCN 2024).

`spec_duran_targets()` Specify targets following Durán *et al.* (2020).

`spec_jung_targets()` Specify targets following Jung *et al.* (2021).

`spec_polak_targets()` Specify targets following Polak *et al.* (2015).

`spec_rodrigues_targets()` Specify targets following Rodrigues *et al.* (2004).

`spec_ward_targets()` Specify targets following Ward *et al.* (2025).

`spec_wilson_targets()` Specify targets following Wilson *et al.* (2010).

`spec_watson_targets()` Specify targets following Watson *et al.* (2010).

`spec_min_targets()` Specify targets based on the minimum of other target setting methods.

`spec_max_targets()` Specify targets based on the maximum of other target setting methods.

- The `add_group_targets()` function provides a convenient interface for adding targets to a conservation planning problem based on groups. By assigning each feature to a group and then specifying a target setting method for each group, it can be used to assign targets for features based on different target setting methods. This function is designed to be used in conjunction with the previously described functions specifying targets for the `add_auto_targets()` function (e.g., `spec_relative_targets()`, `spec_absolute_targets()`, `spec_jung_targets()`). Note that this function is specifically designed for problems that have a single zone, and cannot be used for problems that have multiple zones. For example, if the problem `x` had three features – with the first feature corresponding to an ecosystem and the latter two to different species – this function could be used to set targets for the ecosystem feature based on Polak *et al.* (2015) and targets for the species features based on Jung *et al.* (2021) with the following code.

```
# specify target setting methods for groups with default parameters
# via method names
x %>% add_group_targets(
```

```
  group = c("eco", "spp", "spp"),
  list(eco = "polak", spp = "jung")
)

# specify target setting methods for groups with default parameters
# via functions
x %>% add_group_targets(
  group = c("eco", "spp", "spp"),
  list(eco = spec_polak_targets(), spp = spec_jung_targets())
)

# specify target setting methods for groups with customized parameters
# via functions
x %>% add_group_targets(
  group = c("eco", "spp", "spp"),
  list(
    eco = spec_polak_targets(),
    spp = spec_jung_targets(prop_uplift = 0.05)
  )
)
```

- A set of additional functions are available for adding targets directly to a conservation planning problem. These functions are especially useful when target thresholds have been precomputed, or when considering problems that have multiple zones. In particular, the following functions are available.

  `add_relative_targets()` Add targets expressed as a proportion (between 0 and 1) of the total amount of each feature. Note that this function provides a convenient alternative to `spec_relative_targets()`.

  `add_absolute_targets()` Add targets expressed as the same values as the underlying feature data (ignoring any specified feature units). Note that this function provides a convenient alternative to `spec_absolute_targets()`.

  `add_manual_targets()` Add targets to a conservation planning problem by manually specifying all the required information for each target. This function is especially useful for problems that have multiple zones because it can be used to specify, for a given feature, which zone – or combination of zones – should be considered for assessing feature representation. In addition to specifying targets where feature representation should ideally be greater than or equal to target thresholds (i.e., as is the case for all other target setting functions), this function provides the functionality to specify targets where feature representation should be smaller than or equal to – or equal to – target thresholds (i.e., the constraint sense for the target).

References

Durán AP, Green JMH, West CD, Visconti P, Burgess ND, Virah-Sawmy M, Balmford A (2020) A practical approach to measuring the biodiversity impacts of land conversion. *Methods in Ecology and Evolution*, 11:910–921.

Harris LR, Holness SD (2023) A practical approach to setting heuristic marine biodiversity targets for systematic conservation planning. *Biological Conservation*, 285: 110218.

Jung M, Arnell A, de Lamo X, García-Rangel S, Lewis M, Mark J, Merow C, Miles L, Ondo I, Pironon S, Ravilious C, Rivers M, Schepaschenko D, Tallowin O, van Soesbergen A, Govaerts R, Boyle BL, Enquist BJ, Feng X, Gallagher R, Maitner B, Meiri S, Mulligan M, Ofer G, Roll U, Hanson JO, Jetz W, Di Marco M, McGowan J, Rinnan DS, Sachs JD, Lesiv M, Adams VM, Andrew SC, Burger JR, Hannah L, Marquet PA, McCarthy JK, Morueta-Holme N, Newman EA, Park DS, Roehrdanz PR, Svenning J-C, Violle C, Wieringa JJ, Wynne G, Fritz S, Strassburg BBN, Obersteiner M, Kapos V, Burgess N, Schmidt- Traub G, Visconti P (2021) Areas of global importance for conserving terrestrial biodiversity, carbon and water. *Nature Ecology and Evolution*, 5:1499–1509.

IUCN (2024). Guidelines for the application of IUCN Red List of Ecosystems Categories and Criteria, Version 2.0. Keith DA, Ferrer-Paris JR, Ghoraba SMM, Henriksen S, Monyeki M, Murray NJ, Nicholson E, Rowland J, Skowno A, Slingsby JA, Storeng AB, Valderrábano M, Zager I (Eds.). Gland, Switzerland: IUCN.

IUCN (2025) The IUCN Red List of Threatened Species. Version 2025-1. Available at https://www.iucnredlist.org. Accessed on 23 July 2025.

Polak T, Watson JEM, Fuller RA, Joseph LN, Martin TG, Possingham HP, Venter O, Carwardine J (2015) Efficient expansion of global protected areas requires simultaneous planning for species and ecosystems. *Royal Society Open Science*, 2: 150107.

Rodrigues ASL, Akçakaya HR, Andelman SJ, Bakarr MI, Boitani L, Brooks TM, Chanson JS, Fishpool LDC, Da Fonseca GAB, Gaston KJ, Hoffmann M, Marquet PA, Pilgrim JD, Pressey RL, Schipper J, Sechrest W, Stuart SN, Underhill LG, Waller RW, Watts MEJ, Yan X (2004) Global gap analysis: priority regions for expanding the global protected-area network. *BioScience*, 54: 1092–1100.

Ward M, Possingham HP, Wintle BA, Woinarski JCZ, Marsh JR, Chapple DG, Lintermans M, Scheele BC, Whiterod NS, Hoskin CJ, Aska B, Yong C, Tulloch A, Stewart R, Watson JEM (2025) The estimated cost of preventing extinction and progressing recovery for Australia's priority threatened species. *Proceedings of the National Academy of Sciences*, 122: e2414985122.

Watson JEM, Evans MC, Carwardine J, Fuller RA, Joseph LN, Segan DB, Taylor MFJ, Fensham RJ, Possingham HP (2010) The capacity of Australia's protected-area system to represent threatened species. *Conservation Biology*,25: 324–332.

Wilson KA, Meijaard E, Drummond S, Grantham HS, Boitani L, Catullo G, Christie L, Dennis R, Dutton I, Falcucci A, Maiorano L, Possingham HP, Rondinini C, Turner WR, Venter O, Watts M (2010) Conserving biodiversity in production landscapes. *Ecological Applications*, 20:1721–1732.

See Also

Other overviews: `constraints`, `decisions`, `importance`, `objectives`, `penalties`, `portfolios`, `solvers`, `summaries`

Examples

```
## Not run:
# load data
sim_pu_raster <- get_sim_pu_raster()
sim_features <- get_sim_features()

# create base problem
p0 <-
```

```
  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 <- p0 %>% add_relative_targets(0.1)

# create problem with added absolute targets
p2 <- p0 %>% add_absolute_targets(3)

# create problem with added manual targets,
# and these targets equate to 10% relative targets (same as p1)
target_data <- data.frame(
  feature = names(sim_features),
  target = 0.1,
  type = "relative"
)
p3 <- p0 %>% add_manual_targets(target_data)

# create problem with added automatic targets,
# and these targets equate to 10% relative targets (same as p1)
p4 <- p0 %>% add_auto_targets(method = spec_relative_targets(0.1))

# create problem with added group targets,
# wherein the first 3 features are assigned to group A and have
# 10% targets and the remaining features are assigned to group B and
# have 20% targets
group_data <- c(rep("A", 3), rep("B", terra::nlyr(sim_features) - 3))
p5 <-
   p0 %>%
   add_group_targets(
     groups = group_data,
     method = list(
       "A" = spec_relative_targets(0.1),
       "B" = spec_relative_targets(0.2)
     )
   )

# solve problems
s <- c(solve(p1), solve(p2), solve(p3), solve(p4), solve(p5))
names(s) <- c(
  "relative targets", "absolute targets", "manual targets",
  "automatic targets", "group targets"
)

# plot solutions
plot(s, axes = FALSE)

## 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 `integer` number of rows.

ncol `integer` number of columns.

as.list convert to a `list`.

print print the object.

Examples

```
# load tibble package
require(tibble)

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

# number of rows
nrow(a)

# number of columns
ncol(a)

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

Weight-class *Weight class*

Description

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

Super class

`prioritizr::ConservationModifier` -> `Weight`

Methods

Public methods:

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

Method `output()`**:** Output the weights.

Usage:

```
Weight$output()
```

Returns: `numeric` vector.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other classes: `ConservationModifier-class`, `ConservationProblem-class`, `Constraint-class`, `Decision-class`, `Objective-class`, `OptimizationProblem-class`, `Penalty-class`, `Portfolio-class`, `Solver-class`, `Target-class`, `TargetMethod-class`

`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 either the **Rsymphony** or **gurobi** package to be installed.

Usage

```
write_problem(x, path, solver = NULL)
```

Arguments

`x`	`problem()` 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). If using the Gurobi solver (i.e., `solver = "gurobi"`), the problem can be saved in a compressed file format (e.g., `".mps.gx"`) to reduce file size (see `?gurobi::gurobi_write()` for details).
`solver`	`character` name of optimization solver to write the problem to disk. Available options include: `"rsymphony"` , `"gurobi"`, or `NULL`. Note that using the Gurobi solver is much faster, because the Rsymphony solver requires attempting to solve the problem before it can be written. Defaults to `NULL`, such that the best available solver is used.

Value

An invisible `TRUE` indicating success.

Examples

```
## Not run:
# set seed for reproducibility
set.seed(500)

# load data
sim_pu_polygons <- get_sim_pu_polygons()
sim_features <- get_sim_features()

# subset data to extract first four planning units
sim_pu_polygons <- sim_pu_polygons[1:4, ]

# create minimal problem
p <-
  problem(sim_pu_polygons, sim_features, cost_column = "cost") %>%
```

```
  add_min_set_objective() %>%
  add_relative_targets(0.1) %>%
  add_binary_decisions()

# specify file path to save problem formulation
path <- file.path(tempdir(), "model.lp")
print(path)

# save problem to file
## note that either the gurobi or Rsymphony package needs to be installed
write_problem(p, path)

# print model file
cat(readLines(path), sep = "\n")

## End(Not run)
```

`zones` *Management zones*

Description

Organize data for multiple features for multiple management zones. Specifically, the data should describe the expected amount of each feature within each planning unit given each management zone. For example, the data could describe the occupancy (e.g., presence/absence), probability of occurrence, or abundance expected for each feature when each planning unit is allocated to a different zone.

Usage

```
zones(..., zone_names = NULL, feature_names = NULL)
```

Arguments

`...`	`terra::rast()` 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()`.

`problem()` **will have** `terra::rast()` **or** `sf::st_sf()` **planning units** `terra::rast()` object can be supplied to specify the expected amount of each feature within each planning unit under each management zone. Data for each zone should be specified as separate arguments, and the data for each feature in a given zone are specified in separate layers in a `terra::rast()` object. Note that all layers for a given zone must have missing (`NA`) values in exactly the same cells.

`problem()` **will have** `sf::st_sf()` **or** `data.frame` **planning units** `character` vector containing column names can be supplied to specify the expected amount of each feature under each zone. Note that these columns must not contain any missing (`NA`) values.

`problem()` **will have** `sf::st_sf()`**,** `data.frame`**, or** `matrix` **planning units** `data.frame` object can be supplied to specify the expected amount of each feature under each zone. Following conventions used in *Marxan*, the `data.frame` object should contain the following 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 should correspond to a single zone. Also, note that data are not required for all combinations of planning units, features, and zones. The expected amount of features in planning units under management zones that are missing from the table are assumed to be zero.

Value

A `Zones` object containing data for each zone, and the feature and zone names.

See Also

See `problem()` for information on using this function to generate a prioritization with multiple management zones.

Examples

```
## Not run:
# load planning unit data
sim_pu_raster <- get_sim_pu_raster()

zone_1 <- simulate_species(sim_pu_raster, 3)
zone_2 <- simulate_species(sim_pu_raster, 3)

# create zones using two SpatRaster 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)
```

```
# plot the rasters for the first zone in the Zones object
plot(
 z[[1]],
 main = c("Zone 1 feature 1", "Zone 1 feature 2", "Zone 1 feature 3")
)

# 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 corresponding to column names
# of 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)

## End(Not run)
```

`zone_names` *Zone names*

Description

Extract the names of zones in an object.

Usage

```
zone_names(x, ...)

## S3 method for class 'ConservationProblem'
zone_names(x, ...)

## S3 method for class 'ZonesRaster'
zone_names(x, ...)

## S3 method for class 'ZonesSpatRaster'
zone_names(x, ...)
```

```
## S3 method for class 'ZonesCharacter'
zone_names(x, ...)
```

Arguments

x	problem() or zones() object.
...	not used.

Value

A character vector of zone names.

Examples

```
## Not run:
# load data
sim_zones_pu_raster <- get_sim_zones_pu_raster()
sim_zones_features <- get_sim_zones_features()

# print names of zones in a Zones object
print(zone_names(sim_zones_features))
# create problem with multiple zones
p <-
  problem(sim_zones_pu_raster, sim_zones_features) %>%
  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))

## End(Not run)
```

Index