Package: multiscape (via r-universe)

October 26, 2026

Type Package

Version 1.4.4

Title Multi-Action, Multi-Objective Spatial Planning

Description Provides a modular framework for exact multi-objective spatial planning using mixed-integer programming. The package supports the definition of planning problems through planning units, features, management actions, action effects, spatial relations, targets, constraints, and objective functions. It enables the optimisation of spatial planning portfolios under considerations such as boundary structure, connectivity, and fragmentation. Supported multi-objective methods include weighted-sum aggregation, epsilon-constraint, and the augmented epsilon-constraint method. Problems can be solved with several commercial and open-source optimisation solvers. Optional solver backends include the 'gurobi' R package, which is distributed with the Gurobi Optimizer installation <https://docs.gurobi.com/projects/optimizer/en/13.0/reference/r/setup.html>, and the 'rcbc' R package, available from GitHub at <https://github.com/dirkschumacher/rcbc>. For background on multi-objective optimisation methods, see Halffmann et al. (2022) <doi:10.1002/mcda.1780>; for the augmented epsilon-constraint method, see Mavrotas (2009) <doi:10.1016/j.amc.2009.03.037>.

Depends R (>= 4.1.0)

Imports assertthat (>= 0.2.0), Matrix, proto, magrittr, dplyr, Rcpp, cli, sf, methods, RANN, exactextractr, ggplot2, terra, ggrepel, lifecycle, moocore

Suggests roxygen2, prioritizr, Rsymphony (>= 0.1-31), Rcplex, slam, rlang, testthat (>= 3.0.0), withr, raster, tmap, sp, viridis, markdown, data.table, purrr, readr, tibble, reshape2, rcbc, gurobi, knitr, rmarkdown

LinkingTo Rcpp, RcppArmadillo (>= 0.10.1.0.0), BH

Encoding UTF-8

LazyData true

License GPL (>= 3)

Language en-US

RoxygenNote 7.3.3

URL https://josesalgr.github.io/multiscape/

BugReports https://github.com/josesalgr/multiscape/issues

Collate 'RcppExports.R' 'internal.R' 'add_action_max_per_pu.R'
'add_action_sets.R' 'add_actions.R'
'add_constraint_action_cardinality.R'
'add_constraint_action_relations.R' 'add_constraint_area.R'
'add_constraint_budget.R' 'add_constraint_locked_actions.R'
'add_constraint_locked_planning_units.R'
'add_constraint_targets.R' 'add_effects.R' 'add_objectives.R'
'add_profit.R' 'add_spatial_relations.R' 'internalMO.R'
'analysis_frontier.R' 'analysis_get.R' 'analysis_linkage.R'
'analysis_selection.R' 'analysis_solution.R' 'build_model.R'
'compile_model.R' 'create_problem.R' 'data-sim.R' 'globals.R'
'joint_effects.R' 'load_sim.R' 'numeric_conditioning.R'
'package.R' 'plot.R' 'print.R' 'problem-class.R'
'set_method_augmecon.R' 'set_method_epsilon_constraint.R'
'set_method_weighted_sum.R' 'set_run_design.R' 'set_solver.R'
'solutionset-class.R' 'solution-class.R' 'show.R' 'solve.R'
'utils-pipe.R' 'zzz.R'

Roxygen list(markdown = TRUE)

Config/testthat/edition 3

VignetteBuilder knitr, rmarkdown

Config/pak/sysreqs libabsl-dev cmake libgdal-dev gdal-bin libgeos-dev make libssl-dev libproj-dev libsqlite3-dev libudunits2-dev

Repository https://josesalgr.r-universe.dev

Date/Publication 2026-10-26 12:03:20 UTC

RemoteUrl https://github.com/josesalgr/multiscape

RemoteRef HEAD

RemoteSha 00bc6c00d9c57b8e20b04cf062c6a356849ea7f5

Contents

add_action_sets 4
add_actions 6
add_benefits 11
add_constraint_action_cardinality 12
add_constraint_action_excludes 16
add_constraint_action_requires 18
add_constraint_action_together 21
add_constraint_area 23

add_constraint_budget . . . 27
add_constraint_locked_actions . . . 30
add_constraint_locked_planning_units . . . 33
add_constraint_locked_pu . . . 35
add_constraint_targets_absolute . . . 36
add_constraint_targets_relative . . . 39
add_effects . . . 42
add_losses . . . 48
add_objective_max_benefit . . . 49
add_objective_max_effect . . . 50
add_objective_max_net_profit . . . 52
add_objective_max_profit . . . 54
add_objective_min_cost . . . 56
add_objective_min_effect . . . 59
add_objective_min_fragmentation_action . . . 61
add_objective_min_fragmentation_planning_units . . . 63
add_objective_min_fragmentation_pu . . . 65
add_objective_min_intervention_impact . . . 66
add_objective_min_loss . . . 68
add_profit . . . 70
add_spatial_boundary . . . 73
add_spatial_distance . . . 76
add_spatial_knn . . . 78
add_spatial_queen . . . 80
add_spatial_relations . . . 81
add_spatial_rook . . . 83
compile_model . . . 85
create_problem . . . 86
frontier_distances . . . 91
frontier_extremes . . . 95
frontier_knee . . . 97
frontier_neighbors . . . 100
get_action_sets . . . 102
get_actions . . . 103
get_features . . . 105
get_objectives . . . 107
get_planning_units . . . 109
get_pu . . . 111
get_runs . . . 112
get_solution_states . . . 113
get_targets . . . 115
linkage_contrasts . . . 117
linkage_distances . . . 119
linkage_transition . . . 121
linkage_turnover . . . 124
load_ecosystem_services . . . 126
load_meseta . . . 127
load_sim_features_raster . . . 128

load_sim_multiaction . . . 128
mo_control . . . 129
plot_spatial_actions . . . 130
plot_spatial_features . . . 132
plot_spatial_planning_units . . . 135
plot_spatial_pu . . . 137
plot_tradeoff . . . 138
problem-class . . . 141
run_grid . . . 144
run_manual . . . 145
selection_consistency . . . 145
selection_frequency . . . 147
selection_similarity . . . 149
set_method_augmecon . . . 152
set_method_epsilon_constraint . . . 156
set_method_weighted_sum . . . 159
set_runs_control . . . 162
set_runs_grid . . . 164
set_runs_manual . . . 167
set_solver . . . 169
set_solver_cbc . . . 173
set_solver_cplex . . . 174
set_solver_gurobi . . . 176
set_solver_symphony . . . 178
sim_dist_features . . . 180
sim_features . . . 180
sim_multiaction . . . 180
sim_pu . . . 181
sim_pu_sf . . . 182
solution_append . . . 182
solution_filter . . . 184
solution_unique . . . 188
solutionset-class . . . 191
solve . . . 194

Index **198**

`add_action_sets` *Register sets of management actions*

Description

Define named combinations of existing actions. An action can belong to more than one set. Sets are stored separately from the individual action catalog.

Usage

```
add_action_sets(x, sets)
```

Arguments

x	A Problem object with registered actions.
sets	A non-empty named list of action-id vectors, or a non-empty data.frame with columns set and action. Set identifiers are strings; members can be strings or numeric action ids. Factors are accepted.

Details

Run add_actions() first. Supply either a named list of action-id vectors or a long table with one row per membership and columns set and action. Additional table columns are ignored. Each set must contain at least two distinct registered actions; duplicate memberships and nested sets are rejected. Members are matched by action id, not by their display name.

Set identifiers cannot coincide with action ids, action display names, or existing labels in actions$action_set. The latter column remains an independent, backward-compatible action classification.

Repeated calls add new sets. Reusing a registered set identifier is an error, including attempts to extend or redefine its membership. Definitions are returned in a consistent order by get_action_sets().

Registering a set does not create a selectable action, change feasible planning-unit/action pairs, require its members to be selected together, or add an interaction effect. It also does not enable simultaneous actions: the default maximum of one selected action per planning unit still applies unless changed with add_constraint_action_cardinality(). Sets may be registered even if their members have no common feasible unit. The modern add_effects() interface accepts these identifiers for supplied total joint effects. Objectives and constraints still receive individual members. Joint effects do not make a set a separate selectable action.

Value

A new Problem object containing the definitions. The input object is not modified. Any previously compiled model is invalidated, following the usual problem-construction workflow.

See Also

add_actions(), get_action_sets(), add_constraint_action_cardinality()

Examples

```
problem <- create_problem(
  pu = data.frame(id = 1:2, cost = c(1, 2)),
  features = data.frame(id = 1L, name = "woodland"),
  dist_features = data.frame(
    pu = 1:2, feature = 1L, amount = c(10, 20)
  )
) |>
  add_actions(
    actions = data.frame(id = c("restore", "control", "fence")),
    cost = 1
  )

# An action may belong to several sets.
list_problem <- problem |>
```

```
  add_action_sets(list(
    restore_control = c("restore", "control"),
    restore_fence = c("restore", "fence")
  ))
get_action_sets(list_problem)

# Equivalent definitions in long-table format.
memberships <- data.frame(
  set = c("restore_control", "restore_control", "restore_fence", "restore_fence"),
  action = c("restore", "control", "restore", "fence")
)
table_problem <- add_action_sets(problem, memberships)
identical(get_action_sets(list_problem), get_action_sets(table_problem))

# Add another set without changing existing definitions.
table_problem <- table_problem |>
  add_action_sets(list(control_fence = c("control", "fence")))
get_action_sets(table_problem)
```

`add_actions` *Add management actions to a planning problem*

Description

Define the action catalogue, the set of feasible planning unit–action pairs, their implementation costs, and the effective area represented by each feasible planning unit–action decision.

This function adds two core components to a `Problem` object. First, it stores the action catalogue. Second, it creates the feasible planning unit–action table, including implementation costs, effective action areas, status codes, and internal indices used by the optimization backend.

Conceptually, if $\mathcal{I}$ is the set of planning units and $\mathcal{A}$ is the set of actions, this function determines which pairs $(i, a) \in \mathcal{I} \times \mathcal{A}$ are feasible decisions and assigns a non-negative implementation cost to each feasible pair.

Usage

```
add_actions(
  x,
  actions,
  include_pairs = NULL,
  exclude_pairs = NULL,
  cost = NULL,
  action_area = NULL
)
```

Arguments

x	A Problem object created with create_problem.
actions	A data.frame defining the action catalogue. It must contain a unique id column. A column named action is also accepted and automatically renamed to id.
include_pairs	Optional specification of feasible (pu, action) pairs. It can be NULL, a data.frame with columns pu and action (optionally also feasible), or a named list whose names are action ids and whose elements are vectors of planning unit ids or sf objects.
exclude_pairs	Optional specification of infeasible (pu, action) pairs. It uses the same formats as include_pairs and removes matching pairs from the feasible set.
cost	Optional cost specification for feasible pairs. It may be NULL, a scalar numeric value, a named numeric vector indexed by action id, or a data.frame with columns action, cost or pu, action, cost.
action_area	Optional effective area specification for feasible (pu, action) pairs. It may be NULL or a data.frame with columns pu, action, and action_area. If NULL, action areas are derived from spatial include_pairs when available, otherwise they default to full planning-unit areas when these can be derived from the problem.

Details

When to use add_actions()**.**

Use this function when you want to move from a planning problem defined only by planning units and features to a problem in which decisions are explicitly represented as actions applied in planning units.

Action catalogue.

The actions argument must be a data.frame with a unique id column identifying each action. If a column named action is supplied instead, it is renamed internally to id. Additional columns are preserved. If no name column is provided, action labels are taken from id. If an action_set column is present, it is also preserved and can later be used to refer to groups of actions. To register combinations with many-to-many membership, use add_action_sets after defining the individual actions. Existing definitions are validated whenever the action catalog is updated.

Actions are stored sorted by id to ensure reproducible internal indexing.

Feasible planning unit–action pairs.

Feasibility is controlled through include_pairs and exclude_pairs.

If include_pairs = NULL, all possible (pu, action) pairs are initially considered feasible, that is, all pairs $(i, a) \in \mathcal{I} \times \mathcal{A}$.

If include_pairs is supplied, only those pairs are retained. If exclude_pairs is also supplied, matching pairs are removed afterwards.

More precisely, let $\mathcal{D}^{\text{inc}}$ denote the set of included planning unit–action pairs and let $\mathcal{D}^{\text{exc}}$ denote the set of excluded pairs.

If include_pairs = NULL, the feasible decision set is:

$$\{(i, a) : i \in \mathcal{I},\ a \in \mathcal{A}\} \setminus \mathcal{D}^{\text{exc}}.$$

If include_pairs is supplied, the feasible decision set is:

$$\mathcal{D}^{\text{inc}} \setminus \mathcal{D}^{\text{exc}}.$$

Both include_pairs and exclude_pairs can be specified as:

- NULL,
- a data.frame with columns pu and action,
- or a named list whose names are action ids.

When supplied as a data.frame, the object must contain columns pu and action. An optional logical-like column feasible may also be provided; only rows with feasible = TRUE are retained. Missing values in feasible are treated as FALSE.

When supplied as a named list, names must match action ids. Each element may contain either:

- a vector of planning-unit ids, or
- an sf object defining the spatial zone where the action is feasible.

Lists may mix vectors of planning-unit ids and sf objects across actions. In the spatial case, feasible planning units are identified using sf::st_intersects() against the stored planning-unit geometry. If include_pairs or exclude_pairs contains sf objects, the problem must contain planning-unit geometry in x$data$pu_sf; otherwise an error is raised.

Spatial exclusions are applied after inclusions. If a planning unit–action pair is included and excluded, the exclusion takes precedence and the pair is removed. Spatial exclusions remove complete (pu, action) pairs; they do not partially subtract area from included geometries.

Action areas.

The action_area column in dist_actions stores the effective area represented by each feasible (pu, action) decision. This is an area, not an action intensity.

If action_area = NULL, action areas are derived automatically:

- when include_pairs is supplied using sf objects, action areas are computed as the area of the spatial intersection between planning-unit geometries and the corresponding action geometries;
- when feasible pairs are supplied without spatial geometries, action areas default to the full planning-unit area when this can be derived from the problem object.

If action_area is supplied as a data.frame, it must contain columns pu, action, and action_area. A column named area is also accepted and renamed internally to action_area. Supplied areas must be finite and non-negative. User-supplied values override automatically derived values for matching feasible pairs. Rows referring to non-feasible pairs are ignored with a warning.

If full planning-unit areas cannot be derived for some feasible pairs, action_area is left as NA for those pairs. Area-based action constraints should check for missing action_area values before model construction.

Feasibility versus decision fixing.

This function only determines whether a pair (i, a) exists in the model. It does not force a feasible action to be selected or forbidden beyond structural infeasibility. Fixed decisions should instead be imposed later with `add_constraint_locked_actions`.

Costs.

Costs can be supplied in several ways:

- If `cost = NULL`, all feasible pairs receive a default cost of `1`.
- If `cost` is a scalar, that value is assigned to all feasible pairs.
- If `cost` is a named numeric vector, names must match action ids and costs are assigned by action.
- If `cost` is a `data.frame`, it must define either:
  - action-level costs through columns `action` and `cost`, or
  - pair-specific costs through columns `pu`, `action`, and `cost`.

In all cases, costs must be finite and non-negative.

In practice, a scalar cost is useful when all actions cost the same everywhere, a named vector is useful when cost depends only on action type, and a `(pu, action, cost)` table is useful when cost varies by both planning unit and action.

Status values.

Internally, all feasible pairs are initialized with `status = 0`, meaning that the decision is free. If planning units have already been marked as locked out, then all feasible actions in those planning units are assigned `status = 3`. This preserves consistency with planning-unit exclusions already stored in the problem.

Repeated calls.

The action catalogue and feasible pairs can be defined only once per problem. A second call raises an error, even if the input is identical. To compare action catalogues, build separate problems from the object before this call.

After defining actions, typical next steps include adding effects, optional decision-fixing constraints, objectives, and solver settings before calling `solve()`.

Value

An updated `Problem` object with:

`actions` The action catalogue, including a unique integer `internal_id` for each action.

`dist_actions` The feasible planning unit–action table with columns `pu`, `action`, `cost`, `action_area`, `status`, `internal_pu`, and `internal_action`.

`pu index` A mapping from user-supplied planning-unit ids to internal integer ids.

`action index` A mapping from action ids to internal integer ids.

See Also

`create_problem`, `add_action_sets`, `add_constraint_locked_actions`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

p <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
)

actions <- data.frame(
  id = c("conservation", "restoration"),
  name = c("Conservation", "Restoration")
)

# Example 1: all actions feasible in all planning units
p1 <- add_actions(
  x = p,
  actions = actions,
  cost = c(conservation = 5, restoration = 12)
)

print(p1)
utils::head(p1$data$dist_actions)

# Example 2: specify feasible pairs explicitly
include_df <- data.frame(
  pu = c(1, 2, 3, 4),
  action = c("conservation", "conservation", "restoration", "restoration")
)

p2 <- add_actions(
  x = p,
  actions = actions,
  include_pairs = include_df,
  cost = 10
)

p2$data$dist_actions

# Example 3: remove selected pairs after full expansion
exclude_df <- data.frame(
  pu = c(2, 4),
  action = c("restoration", "conservation")
)

p3 <- add_actions(
  x = p,
  actions = actions,
  exclude_pairs = exclude_df,
  cost = c(conservation = 3, restoration = 8)
```

```
)

utils::head(p3$data$dist_actions)

# Example 4: provide action-specific areas manually
action_area <- data.frame(
  pu = c(1, 2, 3, 4),
  action = c("conservation", "conservation", "restoration", "restoration"),
  action_area = c(100, 50, 80, 100)
)

p4 <- add_actions(
  x = p,
  actions = actions,
  include_pairs = include_df,
  action_area = action_area,
  cost = 10
)

p4$data$dist_actions
```

`add_benefits` *Add benefits*

Description

Legacy convenience wrapper around `add_effects` that retains only positive signed changes (`effect > 0`). Positive change does not necessarily imply an ecological benefit. For new code, prefer `add_effects()` to retain signed effects of both directions. Effects can be defined only once per problem.

Usage

```
add_benefits(
  x,
  benefits = NULL,
  effect_type = c("delta", "after"),
  effect_aggregation = c("sum", "mean")
)
```

Arguments

`x`	A `Problem` created by `create_problem` with feasible actions already defined by `add_actions`.
`benefits`	Alias of `effects`, kept for backwards compatibility.
`effect_type`	Deprecated interpretation argument: `"delta"` for signed changes or `"after"` for expected amounts. Legacy multipliers retain their original interpretation; omit for modern tables.

`effect_aggregation`
: Deprecated raster aggregation argument; use `raster_aggregation` instead.

Value

An updated `Problem` object containing:

`dist_effects` The canonical filtered effects table, containing only rows with `benefit > 0`.

`dist_benefit` A backwards-compatible mirror table containing only the benefit component.

See Also

`add_effects`, `add_losses`, `add_objective_max_benefit`

---

add_constraint_action_cardinality

Constrain the number of actions selected per planning unit

Description

Set a minimum, maximum, or exact number of management actions that can be selected in each planning unit. Rules can apply to all units, selected units, all actions, or a specified subset of actions.

Usage

```
add_constraint_action_cardinality(
  x,
  count,
  sense,
  actions = NULL,
  pu = NULL,
  name = NULL
)
```

Arguments

`x`	A `Problem` object with actions registered using `add_actions()`.
`count`	A single finite, non-negative integer giving the number of selected individual actions per planning unit.
`sense`	One of `"min"`, `"max"`, or `"equal"`.
`actions`	Optional subset of action identifiers or existing `actions$action_set` classification labels. `NULL` (default) counts all actions. Identifiers registered through `add_action_sets()` cannot be counted directly; use their individual member actions.
`pu`	Optional vector of planning-unit identifiers. `NULL` (default) applies the rule separately to all planning units.
`name`	Optional name for the constraint. A unique name is generated when `NULL`.

Details

How action cardinality works

A cardinality constraint limits the number of individual actions selected *within each planning unit*. For example, `count = 2` and `sense = "max"` allow zero, one, or two actions per unit, whereas `sense = "equal"` requires exactly two. The available senses are:

- `"max"`: select at most `count` actions.
- `"min"`: select at least `count` actions.
- `"equal"`: select exactly `count` actions.

The limit is evaluated separately for every unit covered by `pu`; it is not a total number of actions across the study area.

Default and overlapping rules

By default, at most one action can be selected per planning unit. To allow concurrent actions, specify a total maximum or exact-count rule with `actions = NULL`. That rule replaces the default one-action limit only in the planning units it covers. A minimum rule or a rule restricted to an action subset does not remove the default limit.

For example, to require at least two actions in a unit, first allow at least two with a total maximum, then add the minimum. Rules from different calls are enforced together; later rules do not overwrite earlier ones. Conflicting rules can make the problem infeasible.

Restricting the scope

Supply `pu` to restrict a rule to particular planning units and `actions` to count only specified actions. For example, a minimum of one action from `c("restore", "control")` requires at least one of those actions in every covered unit. Actions excluded by the subset are not counted towards that particular rule.

Cardinality counts individual selected actions, not the action sets registered with `add_action_sets()`. Registered sets describe joint effects when their member actions are selected; they are not additional selectable actions.

Feasibility and stored rules

Only available planning-unit/action pairs can be selected. A positive minimum or exact count larger than the number of available actions in a covered unit is rejected. Conflicts with other constraints may still be detected only when the problem is solved. Explicit rules are stored in `x$data$constraints$action_cardinality` and do not themselves select actions or run an optimisation solver.

Value

An updated `Problem` with an additional rule in `x$data$constraints$action_cardinality`. The input problem is preserved; no optimisation is performed.

See Also

`add_actions`, `add_action_sets`, `add_constraint_budget`

Examples

```
# EXAMPLE 1: Define the available decisions

# Consider three planning units, each with four possible actions.
# By default, the model allows at most one action in each unit.
# Registering actions does not select any of them.

problem <- create_problem(
  pu = data.frame(id = c(10L, 20L, 30L), cost = 1),
  features = data.frame(id = 1L, name = "woodland"),
  dist_features = data.frame(
    pu = c(10L, 20L, 30L), feature = 1L, amount = 10
  )
) |>
  add_actions(
    data.frame(id = c("restore", "control", "fence", "monitor")),
    cost = 1
  )

# EXAMPLE 2: Different capacities in different planning units

# Allow up to three actions in unit 10 and up to two in unit 20.
# Unit 30 retains the default maximum of one action.
# A maximum permits fewer actions, including no intervention.

capacities <- problem |>
  add_constraint_action_cardinality(
    count = 3, sense = "max", pu = 10L, name = "capacity_10"
  ) |>
  add_constraint_action_cardinality(
    count = 2, sense = "max", pu = 20L, name = "capacity_20"
  )

# Inspect the registered constraints, not solved decisions.
capacities$data$constraints$action_cardinality[
  , c("name", "count", "sense")
]

# EXAMPLE 3: Require at least one action from a subset

# In units 10 and 20, require restoration or control (or both).
# The minimum applies separately to each unit, and only these
# two actions count towards the requirement. The previously
# defined total capacities remain in force.

required <- capacities |>
  add_constraint_action_cardinality(
    count = 1, sense = "min",
    actions = c("restore", "control"),
    pu = c(10L, 20L), name = "management_required"
  )
```

```
# EXAMPLE 4: Require an exact count or prohibit interventions

# An exact count of two overrides the default one-action maximum
# in unit 10 and requires two distinct actions to be selected.

exact <- add_constraint_action_cardinality(
  problem, count = 2, sense = "equal", pu = 10L
)

# A zero maximum prohibits all actions in unit 30.
# It does not exclude that unit from the planning region.

no_action <- add_constraint_action_cardinality(
  problem, count = 0, sense = "max", pu = 30L
)

# EXAMPLE 5: Map where a more flexible capacity is allowed

# Use the bundled 64-unit landscape to impose a hypothetical
# policy allowing two concurrent actions in the eastern half.
# All other planning units retain the default maximum of one.
# No solver is needed to visualise the scope of the rule.

if (requireNamespace("sf", quietly = TRUE)) {
  sim <- load_sim_multiaction()
  spatial_problem <- create_problem(
    pu = sim$planning_units,
    features = sim$features,
    dist_features = sim$dist_features,
    cost = "cost"
  ) |>
    add_actions(sim$actions, cost = sim$action_costs)

  # Identify the eastern units using the x-coordinate of their
  # polygon centroids. The two-action limit is an upper bound,
  # not a requirement to select two actions.
  centres <- sf::st_coordinates(
    sf::st_centroid(sf::st_geometry(sim$planning_units))
  )
  east <- centres[, 1] > stats::median(centres[, 1])
  eastern_ids <- sim$planning_units$id[east]

  spatial_problem <- add_constraint_action_cardinality(
    spatial_problem, count = 2, sense = "max", pu = eastern_ids
  )

  # Display the geographical limits, not selected actions.
  capacity_map <- sim$planning_units
  capacity_map$max_actions <- ifelse(east, 2L, 1L)
  plot(capacity_map["max_actions"], main = "Maximum actions per unit")
}
```

`add_constraint_action_excludes` *Make actions mutually exclusive within each planning unit*

Description

Prevent two or more incompatible actions from being selected together in the same planning unit. At most one action in the specified group may be selected; selecting none is also allowed.

Usage

```
add_constraint_action_excludes(x, actions, pu = NULL, name = NULL)
```

Arguments

`x`	A `Problem` with actions registered by `add_actions()`.
`actions`	At least two action IDs or existing classification labels that must be mutually exclusive in each scoped planning unit.
`pu`	Optional external planning-unit IDs. `NULL` applies the rule to all planning units.
`name`	Optional unique, non-empty constraint label. If omitted, a label is generated automatically.

Details

How the relation works

Suppose restoration and harvesting are incompatible uses of a site. This relation prevents their simultaneous selection without prohibiting either action individually. With three or more actions, **every pair in the group** is mutually exclusive, not just the complete combination.

For every scoped unit, the constraint is $\sum_{a \in A} x_{ia} \leq 1$. Actions outside the group are not restricted by this relation. The rule is most informative when the cardinality settings otherwise permit more than one action per unit.

Value

A new `Problem` with the relation appended. The input problem is unchanged.

Scope and feasibility

Each relation applies separately within each unit specified by `pu`; `pu = NULL` applies it to all registered planning units. Relations do not select actions by themselves or make unavailable actions feasible. **By default, at most one action can be selected per planning unit.** Use `add_constraint_action_cardinality()` when a requires or together rule needs simultaneous selections. Unavailable or locked-out actions are treated as unselected: missing companions may prohibit a trigger, and a missing together member may prohibit the other members. Combined rules, budgets, and locks can also make a model infeasible.

Relations do not imply a temporal sequence or dependencies between different planning units. They can be combined with joint effects registered through add_action_sets(), but defining a joint effect does not itself require actions to be selected together.

Repeated calls

Distinct rules accumulate and are enforced simultaneously. Duplicate rules with the same type, action groups, sense, and PU scope are rejected; names must be unique across requires, excludes, and together relations. To revise a rule, start from the problem before that rule was added.

See Also

add_constraint_action_requires(), add_constraint_action_together(), add_constraint_action_cardinality(

Examples

```
# EXAMPLE 1: Allow multiple actions, but exclude an incompatible pair

# Up to three actions may be selected per unit. Restoration and harvesting
# are incompatible, while threat control can accompany either one.

base <- create_problem(
  pu = data.frame(id = 1:2, cost = 0),
  features = data.frame(id = 1L, name = "habitat"),
  dist_features = data.frame(pu = 1:2, feature = 1L, amount = 100)
) |>
  add_actions(
    data.frame(id = c("restore", "harvest", "control", "monitor")),
    cost = 1
  ) |>
  add_constraint_action_cardinality(3, "max")

exclusive <- add_constraint_action_excludes(
  base,
  actions = c("restore", "harvest"),
  name = "incompatible_uses"
)

# EXAMPLE 2: Restrict the rule to selected planning units

# In unit 1, choosing any one of restoration, harvesting, or control
# precludes the other two. The same rule is not imposed in unit 2.

local <- add_constraint_action_excludes(
  base,
  actions = c("restore", "harvest", "control"),
  pu = 1L,
  name = "local_exclusion"
)

# Inspect the stored relation, not a selected management plan.
local$data$constraints$action_relations[, c("type", "name")]
```

`add_constraint_action_requires` *Require companion actions in each planning unit*

Description

Make the selection of one or more actions conditional on selecting companion actions in the same planning unit. With `sense = "all"`, every companion is required; with `sense = "any"`, at least one is required.

Usage

```
add_constraint_action_requires(
  x,
  actions,
  requires,
  sense = c("all", "any"),
  pu = NULL,
  name = NULL
)
```

Arguments

`x`	A `Problem` with actions registered by `add_actions()`.
`actions`	Non-empty vector of triggering action IDs or existing `actions$action_set` classification labels. For excludes and together, this is the group of related actions and must resolve to at least two distinct actions. Identifiers registered through `add_action_sets()` are not selectable actions; supply their individual members.
`requires`	Non-empty vector of companion action IDs or existing classification labels, disjoint from `actions`.
`sense`	`"all"` (default) requires every companion; `"any"` requires at least one companion.
`pu`	Optional external planning-unit IDs. `NULL` applies the rule to all planning units.
`name`	Optional unique, non-empty constraint label. If omitted, a label is generated automatically.

Details

How the relation works

Suppose restoration is effective only when threat control is also carried out. If restoration is selected, control must also be selected; if restoration is not selected, the relation imposes no requirement. With several triggering actions, each independently activates the requirement.

For binary action decisions x_{ia}, requiring all companions gives $x_{ia} \leq x_{ib}$ for each required action b. Requiring any companion gives $x_{ia} \leq \sum_b x_{ib}$. When there is only one companion, `"all"` and `"any"` are equivalent.

Value

A new `Problem` with the relation appended to `x$data$constraints$action_relations`. The input problem is unchanged.

Scope and feasibility

Each relation applies separately within each unit specified by pu; `pu = NULL` applies it to all registered planning units. Relations do not select actions by themselves or make unavailable actions feasible. **By default, at most one action can be selected per planning unit.** Use `add_constraint_action_cardinality()` when a requires or together rule needs simultaneous selections. Unavailable or locked-out actions are treated as unselected: missing companions may prohibit a trigger, and a missing together member may prohibit the other members. Combined rules, budgets, and locks can also make a model infeasible.

Relations do not imply a temporal sequence or dependencies between different planning units. They can be combined with joint effects registered through `add_action_sets()`, but defining a joint effect does not itself require actions to be selected together.

Repeated calls

Distinct rules accumulate and are enforced simultaneously. Duplicate rules with the same type, action groups, sense, and PU scope are rejected; names must be unique across requires, excludes, and together relations. To revise a rule, start from the problem before that rule was added.

See Also

`add_constraint_action_excludes()`, `add_constraint_action_together()`, `add_constraint_action_cardinality()`, `add_action_sets()`

Examples

```
# EXAMPLE 1: Create a problem where concurrent actions are possible

# Three planning units offer restoration, control, fencing, and monitoring.
# Allow up to three actions per unit: the default one-action maximum
# would otherwise prevent restoration and its companions being selected.

base <- create_problem(
  pu = data.frame(id = c(10L, 20L, 30L), cost = 0),
  features = data.frame(id = 1L, name = "habitat"),
  dist_features = data.frame(
    pu = c(10L, 20L, 30L), feature = 1L, amount = 100
  )
) |>
  add_actions(
    data.frame(id = c("restore", "control", "fence", "monitor")),
    cost = 1
```

```
  ) |>
  add_constraint_action_cardinality(3, "max")

# EXAMPLE 2: Require all companions in a specific unit

# In unit 10, restoration must be accompanied by both control and fencing.
# This is a conditional requirement: it does not force restoration.

required <- add_constraint_action_requires(
  base,
  actions = "restore",
  requires = c("control", "fence"),
  sense = "all",
  pu = 10L,
  name = "restore_with_both"
)

# EXAMPLE 3: Require at least one alternative companion

# In unit 20, either control OR fencing is sufficient for restoration.
# These rules have different PU scopes and can coexist.

required <- add_constraint_action_requires(
  required,
  actions = "restore",
  requires = c("control", "fence"),
  sense = "any",
  pu = 20L,
  name = "restore_with_either"
)

# EXAMPLE 4: Multiple independent triggers

# In unit 30, choosing either restoration or monitoring requires control.
# The triggers need not both be selected.

required <- add_constraint_action_requires(
  required,
  actions = c("restore", "monitor"),
  requires = "control",
  pu = 30L
)

# Inspect the three registered rules (not an optimised solution).
required$data$constraints$action_relations[, c("type", "sense", "name")]

# EXAMPLE 5: Map where a requirement applies (no solver needed)

if (requireNamespace("sf", quietly = TRUE)) {
  sim <- load_sim_multiaction()

  # Apply the restoration-protection requirement to the first 16 cells.
  # Both actions may be selected in these cells, so permit cardinality 2.
```

```
  scoped_pu <- head(sim$planning_units$id, 16)
  spatial <- create_problem(
    pu = sim$planning_units,
    features = sim$features,
    dist_features = sim$dist_features,
    cost = "cost"
  ) |>
    add_actions(sim$actions, cost = sim$action_costs) |>
    add_constraint_action_cardinality(2, "max", pu = scoped_pu) |>
    add_constraint_action_requires(
      "restore", "protect", pu = scoped_pu
    )

  # The map shows the SCOPE of the requirement, not selected actions.
  mapped <- sim$planning_units
  mapped$requirement <- ifelse(
    mapped$id %in% scoped_pu, "Applies", "Does not apply"
  )
  plot(mapped["requirement"], main = "Restoration requires protection")
}
```

`add_constraint_action_together` *Select actions together within each planning unit*

Description

Link a group of actions so that, in each specified planning unit, either all are selected or none are selected.

Usage

```
add_constraint_action_together(x, actions, pu = NULL, name = NULL)
```

Arguments

`x`	A `Problem` with actions registered by `add_actions()`.
`actions`	At least two action IDs or existing classification labels that must be selected together or omitted together in each scoped unit.
`pu`	Optional external planning-unit IDs. `NULL` applies the rule to all planning units.
`name`	Optional unique, non-empty constraint label. If omitted, a label is generated automatically.

Details

How the relation works

Suppose restoration and threat control must be implemented as one package. This relation requires both actions whenever either is selected. It does **not** require either action to be selected in the first place.

The binary decisions satisfy $x_{ia} = x_{ib}$ for all members of the group. Unlike `add_constraint_action_requires()`, dependency is bidirectional. Unlike `add_constraint_action_excludes()`, simultaneous selection is permitted and, when the group is used, required.

Cardinality must allow all members to be selected. Registering a set with `add_action_sets()` specifies a possible combination for joint effects; it does not itself impose the together relation.

Value

A new `Problem` with the relation appended. The input problem is unchanged.

Scope and feasibility

Each relation applies separately within each unit specified by `pu`; `pu = NULL` applies it to all registered planning units. Relations do not select actions by themselves or make unavailable actions feasible. **By default, at most one action can be selected per planning unit.** Use `add_constraint_action_cardinality()` when a requires or together rule needs simultaneous selections. Unavailable or locked-out actions are treated as unselected: missing companions may prohibit a trigger, and a missing together member may prohibit the other members. Combined rules, budgets, and locks can also make a model infeasible.

Relations do not imply a temporal sequence or dependencies between different planning units. They can be combined with joint effects registered through `add_action_sets()`, but defining a joint effect does not itself require actions to be selected together.

Repeated calls

Distinct rules accumulate and are enforced simultaneously. Duplicate rules with the same type, action groups, sense, and PU scope are rejected; names must be unique across requires, excludes, and together relations. To revise a rule, start from the problem before that rule was added.

See Also

`add_constraint_action_requires()`, `add_constraint_action_excludes()`, `add_constraint_action_cardinality()`, `add_action_sets()`

Examples

```
# EXAMPLE 1: Require restoration and control to occur together

# In planning unit 1, restoration and control must be selected together
# or both omitted. Monitoring remains independently selectable.
# Allow two simultaneous actions in unit 1; elsewhere the default
# maximum of one action still applies.
```

```
base <- create_problem(
  pu = data.frame(id = 1:2, cost = 0),
  features = data.frame(id = 1L, name = "habitat"),
  dist_features = data.frame(pu = 1:2, feature = 1L, amount = 100)
) |>
  add_actions(
    data.frame(id = c("restore", "control", "monitor")), cost = 1
  ) |>
  add_constraint_action_cardinality(2, "max", pu = 1L)

together <- add_constraint_action_together(
  base,
  actions = c("restore", "control"),
  pu = 1L,
  name = "restoration_package"
)

# Inspect the registered requirement; no solver has been run.
together$data$constraints$action_relations[, c("type", "name")]

# EXAMPLE 2: Extend the package to three actions

# All three must be selected or omitted as a group. The cardinality
# upper bound must now allow three actions in the scoped unit.

extended <- create_problem(
  pu = data.frame(id = 1L, cost = 0),
  features = data.frame(id = 1L, name = "habitat"),
  dist_features = data.frame(pu = 1L, feature = 1L, amount = 100)
) |>
  add_actions(
    data.frame(id = c("restore", "control", "monitor")), cost = 1
  ) |>
  add_constraint_action_cardinality(3, "max") |>
  add_constraint_action_together(c("restore", "control", "monitor"))
```

`add_constraint_area` *Add area constraint*

Description

Add an area constraint to a planning problem.

This function stores one area-constraint specification in the `Problem` object so that it can later be incorporated when the optimization model is assembled. Area constraints can be imposed either on the total selected planning-unit area or on the effective area of selected actions.

Multiple area constraints can be added by calling this function repeatedly, provided that no duplicated combination of action subset and constraint sense is introduced.

Usage

```
add_constraint_area(
  x,
  area,
  sense,
  tolerance = 0,
  area_col = NULL,
  area_unit = c("m2", "ha", "km2"),
  actions = NULL,
  name = NULL
)
```

Arguments

x	A Problem object.
area	Numeric scalar greater than or equal to zero. Target value for the constrained area.
sense	Character string indicating the type of area constraint. Must be one of "min", "max", or "equal".
tolerance	Numeric scalar greater than or equal to zero. Only used when sense = "equal". In that case, equality is interpreted as a band around area with half-width tolerance. Ignored otherwise.
area_col	Optional character string giving the name of the planning-unit area column. When actions = NULL, this column is used as the source of planning-unit areas. When actions is not NULL, the constraint uses x$data$dist_actions$action_area; area_col can still be used by the model builder as a fallback if action areas are missing and full planning-unit areas can be derived.
area_unit	Character string indicating the unit of area and tolerance. Must be one of "m2", "ha", or "km2".
actions	Optional subset of actions to which the constraint applies. If NULL, the constraint applies to the total selected planning-unit area through the planning-unit selection variables. Otherwise, it applies to the effective area of the selected decision variables associated with the specified subset of actions, using x$data$dist_actions$action_area. This argument is resolved using the package's standard action subset parser.
name	Optional character string used as the label of the stored linear constraint when it is later added to the optimization model. If NULL, a default name is generated.

Details

Calls accumulate constraints for distinct action scopes, area measures (area_col), or senses. Repeating the same combination raises an error, even with a different threshold, display unit, or name. To change it, rebuild from the problem before the constraint was added. Explicit names must be unique within the area-constraint family; omitted names are generated automatically.

Use this function when area requirements must be imposed either on the total selected landscape or on the effective area allocated to specific actions.

Total selected area.

Let $\mathcal{I}$ denote the set of planning units and let $a_i \geq 0$ be the area associated with planning unit $i \in \mathcal{I}$.

When `actions = NULL`, the constraint refers to the total selected area in the problem. In that case, let $w_i \in \{0, 1\}$ denote the binary variable indicating whether planning unit i is selected by at least one decision in the model.

Depending on `sense`, this function stores one of the following constraints.

If `sense = "min"`:

$$\sum_{i \in \mathcal{I}} a_i w_i \geq A$$

If `sense = "max"`:

$$\sum_{i \in \mathcal{I}} a_i w_i \leq A$$

If `sense = "equal"` and `tolerance = 0`:

$$\sum_{i \in \mathcal{I}} a_i w_i = A$$

If `sense = "equal"` and `tolerance > 0`, the equality is stored as a two-sided band:

$$A - \tau \leq \sum_{i \in \mathcal{I}} a_i w_i \leq A + \tau$$

where τ is the value supplied through `tolerance`.

Action-specific effective area.

When `actions` is not `NULL`, the constraint refers to the effective area of the selected decisions associated with the specified subset of actions.

Let $\mathcal{A}^\star \subseteq \mathcal{A}$ denote the specified action subset and let $x_{ia} \in \{0, 1\}$ denote the binary variable indicating whether action $a \in \mathcal{A}^\star$ is selected in planning unit $i \in \mathcal{I}$. Let $b_{ia} \geq 0$ denote the effective action area associated with feasible pair (i, a), as stored in `x$data$dist_actions$action_area`. In that case, the constrained quantity is:

$$\sum_{i \in \mathcal{I}} \sum_{a \in \mathcal{A}^\star} b_{ia} x_{ia}.$$

Effective action areas are defined in `add_actions`. They can be supplied manually through the `action_area` argument, derived from spatial `include_pairs`, or default to full planning-unit areas when these can be derived from the problem.

Consequently, if no partial action areas are supplied and each feasible action is assumed to apply to the full planning unit, then an action-specific area constraint is numerically equivalent to using planning-unit areas for those action decisions. If partial action areas are supplied, the constraint uses those effective areas instead.

Area units and area sources.

The value of `area_unit` indicates the unit in which `area` and `tolerance` are expressed.

When `actions = NULL`, areas are obtained from the planning-unit table or planning-unit geometry. If `area_col` is provided, that column is used. Otherwise, the model builder determines the default area source according to the internal rules of the package.

When `actions` is not `NULL`, the model builder uses `x$data$dist_actions$action_area`. These values are stored internally in square metres and are converted to `area_unit` when the model is built. If action areas are missing for relevant feasible `(pu, action)` pairs, the model builder may use `area_col` as a fallback to derive full planning-unit areas. If valid action areas still cannot be obtained, model construction fails with an error.

This function only stores the constraint specification; it does not validate the feasibility of the threshold against the available planning units or actions at this stage.

Multiple area constraints can be stored in a `Problem` object. However, at most one can be stored for the same combination of action subset and constraint sense. Attempting to add a duplicated `actions`–`sense` combination results in an error.

Value

An updated `Problem` object with the new area-constraint specification appended to `x$data$constraints$area`.

See Also

`create_problem`, `add_actions`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

p <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  )

# Constrain the total selected planning-unit area.
p <- add_constraint_area(
  x = p,
  area = 25,
  sense = "min",
  area_unit = "ha"
)

# Constrain the effective area allocated to the restore action.
# Action areas are derived from the planning-unit geometries.

p <- add_constraint_area(
  x = p,
  area = 15,
  sense = "max",
  area_unit = "ha",
```

```
  actions = "restore"
)

p$data$constraints$area
```

`add_constraint_budget` *Add budget constraint*

Description

Add a budget constraint to a planning problem.

This function stores one budget-constraint specification in the `Problem` object so that it can later be incorporated when the optimization model is assembled. Multiple budget constraints can be added by calling this function repeatedly, provided that no duplicated combination of action subset and constraint sense is introduced.

Usage

```
add_constraint_budget(
  x,
  budget,
  sense,
  tolerance = 0,
  actions = NULL,
  include_pu_cost = TRUE,
  include_action_cost = TRUE,
  name = NULL
)
```

Arguments

`x`	A `Problem` object.
`budget`	Numeric scalar greater than or equal to zero. Target value for the constrained budget.
`sense`	Character string indicating the type of budget constraint. Must be one of `"min"`, `"max"`, or `"equal"`.
`tolerance`	Numeric scalar greater than or equal to zero. Only used when `sense = "equal"`. In that case, equality is interpreted as a band around `budget` with half-width `tolerance`. Ignored otherwise.
`actions`	Optional subset of actions to which the constraint applies. If `NULL`, the constraint applies to the whole problem. Otherwise, it applies only to the selected decision variables associated with the specified subset of actions. This argument is resolved using the package's standard action subset parser.

include_pu_cost
: Logical scalar indicating whether planning-unit costs should be included in the constrained budget. This is only supported when `actions = NULL`.

include_action_cost
: Logical scalar indicating whether action costs should be included in the constrained budget.

name
: Optional character string used as the label of the stored linear constraint when it is later added to the optimization model. If `NULL`, a default name is generated.

Details

Calls accumulate constraints for distinct action scopes, cost components, or senses. Repeating the same combination raises an error, even with a different threshold or name. To change it, rebuild from the problem before the constraint was added. Explicit names must be unique within the budget-constraint family; omitted names are generated automatically.

Use this function when spending limits or minimum spending requirements must be imposed either on the full problem or on the subset of selected decisions associated with specific actions.

Let $\mathcal{I}$ denote the set of planning units and let $\mathcal{A}$ denote the set of actions. Let $w_i \in \{0, 1\}$ denote the binary variable indicating whether planning unit $i \in \mathcal{I}$ is selected by at least one decision in the model, and let $x_{ia} \in \{0, 1\}$ denote the binary variable indicating whether action $a \in \mathcal{A}$ is selected in planning unit $i \in \mathcal{I}$.

The total constrained budget can include two cost components:

- planning-unit costs, associated with w_i;
- action costs, associated with x_{ia}.

The arguments `include_pu_cost` and `include_action_cost` determine which of these components are included in the stored budget constraint.

When `actions = NULL`, the constraint refers to the total budget across the whole problem. In that case, depending on the values of `include_pu_cost` and `include_action_cost`, the constrained quantity is one of the following:

If only planning-unit costs are included:

$$\sum_{i \in \mathcal{I}} c_i^{pu} w_i$$

If only action costs are included:

$$\sum_{i \in \mathcal{I}} \sum_{a \in \mathcal{A}} c_{ia}^{act} x_{ia}$$

If both components are included:

$$\sum_{i \in \mathcal{I}} c_i^{pu} w_i + \sum_{i \in \mathcal{I}} \sum_{a \in \mathcal{A}} c_{ia}^{act} x_{ia}$$

Depending on `sense`, this function stores one of the following constraints:

If `sense = "min"`:

$$C \geq B$$

If `sense = "max"`:

$$C \leq B$$

If `sense = "equal"` and `tolerance = 0`:

$$C = B$$

If `sense = "equal"` and `tolerance > 0`, the equality is stored as a two-sided band:

$$B - \tau \leq C \leq B + \tau$$

where C denotes the selected cost expression and τ is the value supplied through `tolerance`.

When `actions` is not `NULL`, only action costs can be included. In that case, let $\mathcal{A}^{\star} \subseteq \mathcal{A}$ denote the selected subset of actions, and the constrained quantity is

$$\sum_{i \in \mathcal{I}} \sum_{a \in \mathcal{A}^{\star}} c_{ia}^{act} x_{ia}.$$

Action-specific budget constraints only support action costs. Therefore, `include_pu_cost = TRUE` is only allowed when `actions = NULL`, because planning-unit costs are not action-specific.

This function only stores the constraint specification; it does not validate the feasibility of the threshold against the available cost data at this stage.

Multiple budget constraints can be stored in a `Problem` object. However, at most one can be stored for the same combination of action subset and constraint sense. Attempting to add a duplicated `actions`–`sense` combination results in an error.

Value

An updated `Problem` object with the new budget-constraint specification appended to the stored budget-constraint table.

See Also

`create_problem`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

p <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  )
```

```
p <- add_constraint_budget(
  x = p,
  budget = 10,
  sense = "max",
  include_pu_cost = TRUE,
  include_action_cost = TRUE
)

p <- add_constraint_budget(
  x = p,
  budget = 4,
  sense = "max",
  actions = "restore",
  include_pu_cost = FALSE,
  include_action_cost = TRUE
)

p <- add_constraint_budget(
  x = p,
  budget = 1,
  sense = "min",
  actions = "restore",
  include_pu_cost = FALSE,
  include_action_cost = TRUE
)

p$data$constraints$budget
```

`add_constraint_locked_actions` *Add locked action decisions to a planning problem*

Description

Fix feasible planning unit–action decisions to be selected or excluded.

This function modifies the status of existing feasible `(pu, action)` pairs stored in the feasible action table. It does not create new feasible action pairs and therefore must be used only after `add_actions` has been called.

Usage

```
add_constraint_locked_actions(x, locked_in = NULL, locked_out = NULL)
```

Arguments

`x`	A `Problem` object with action feasibility already defined via `add_actions`.
`locked_in`	Optional specification of feasible `(pu, action)` pairs that must be selected. It may be `NULL`, a `data.frame`, or a named list.

`locked_out` Optional specification of feasible `(pu, action)` pairs that must not be selected. It may be `NULL`, a `data.frame`, or a named list.

Details

Use this function when only specific feasible `(pu, action)` decisions must be forced in or out of the solution, rather than whole planning units.

Let $\mathcal{I}$ denote the set of planning units and $\mathcal{A}$ the set of actions. Let $\mathcal{D} \subseteq \mathcal{I} \times \mathcal{A}$ denote the set of feasible planning unit–action pairs already defined in the problem.

This function allows the user to define two subsets:

- $\mathcal{D}^{in} \subseteq \mathcal{D}$, the set of feasible pairs that must be selected,
- $\mathcal{D}^{out} \subseteq \mathcal{D}$, the set of feasible pairs that must not be selected.

These sets are encoded by updating the `status` column of the feasible action table. The function validates that all requested locked-in and locked-out pairs are already feasible. Therefore, it cannot be used to introduce new planning unit–action combinations into the problem.

In optimization terms, if x_{ia} denotes the decision variable associated with planning unit i and action a, then:

- locked-in pairs conceptually impose $x_{ia} = 1$,
- locked-out pairs conceptually impose $x_{ia} = 0$.

The exact translation into solver-side constraints occurs later when the model is built.

In contrast, `add_constraint_locked_pu` fixes whole planning units through the unit-selection variables, whereas this function fixes only specific feasible `(pu, action)` decisions.

Accepted formats

Both `locked_in` and `locked_out` accept the same formats:

- `NULL`,
- a `data.frame` with columns `pu` and `action`, optionally including a `feasible` column used as a filter,
- a named list whose names are action ids and whose elements are either vectors of planning unit ids or `sf` objects.

If a `feasible` column is supplied in a `data.frame`, only rows with `feasible = TRUE` are used. Missing values in `feasible` are treated as `FALSE`.

If an `sf` specification is supplied, the problem object must contain planning-unit geometry, and planning units are matched spatially using `sf::st_intersects()`.

Conflict checking

Calls accumulate compatible locks and preserve omitted arguments. Repeating the same lock is idempotent. Changing an existing locked-in pair to locked-out, or vice versa, raises an error. To change a lock, rebuild from the problem before it was added.

A given `(pu, action)` pair cannot be simultaneously requested in both `locked_in` and `locked_out`. Such overlaps are rejected.

In addition, if a planning unit is already marked as locked out at the planning-unit level, then all feasible actions in that planning unit are forced to `status = 3`. Any attempt to lock in an action within such a planning unit raises an error.

Order of precedence

User-supplied locked-in and locked-out action requests are first applied to the feasible action table. Afterwards, any planning-unit-level `locked_out` flag is enforced, overriding action-level status and ensuring consistency with planning-unit exclusions.

Value

An updated `Problem` object in which the status column of the feasible action table has been modified to reflect locked-in and locked-out decisions.

See Also

`add_actions`, `add_constraint_locked_pu`

Examples

```
# Build a small spatial planning problem with two available actions.
sim <- load_sim_multiaction()
p <- create_problem(
  pu = sim$planning_units,
  features = sim$features,
  dist_features = sim$dist_features,
  cost = "cost"
) |>
  add_actions(sim$actions, cost = sim$action_costs)

# EXAMPLE 1: Fix individual planning-unit/action decisions.
# Protection must be selected in unit 1, restoration in unit 2,
# and protection cannot be selected in unit 4.
locked <- add_constraint_locked_actions(
  p,
  locked_in = data.frame(
    pu = c(1, 2), action = c("protect", "restore")
  ),
  locked_out = data.frame(pu = 4, action = "protect")
)

# Only show modified decisions: 2 = locked in, 3 = locked out.
# All remaining feasible decisions retain status 0 (free).
subset(locked$data$dist_actions, status %in% c(2L, 3L),
       select = c(pu, action, status))

# EXAMPLE 2: Specify the same locks using a named list.
# List names are action IDs; values are planning-unit IDs.
locked_list <- add_constraint_locked_actions(
  p,
  locked_in = list(protect = 1, restore = 2),
  locked_out = list(protect = 4)
```

```
)

# Both input formats lead to the same decision statuses.
identical(locked$data$dist_actions$status,
          locked_list$data$dist_actions$status)
```

`add_constraint_locked_planning_units` *Add locked planning units to a problem*

Description

Define planning units that must be included in, or excluded from, the optimization problem.

This function updates the planning-unit table stored in the `Problem` object by creating or replacing the logical columns `locked_in` and `locked_out`. These columns are later used by the model builder when translating the problem into optimization constraints.

Lock information may be supplied either directly as logical vectors, as vectors of planning-unit ids, or by referencing columns in the raw planning-unit data originally passed to `create_problem`.

Usage

```
add_constraint_locked_planning_units(x, locked_in = NULL, locked_out = NULL)
```

Arguments

`x`	A `Problem` object created with `create_problem`.
`locked_in`	Optional locked-in specification. It may be `NULL`, a column name in the raw planning-unit data, a logical vector, or a vector of planning-unit ids.
`locked_out`	Optional locked-out specification. It may be `NULL`, a column name in the raw planning-unit data, a logical vector, or a vector of planning-unit ids.

Details

Use this function when whole planning units must be forced into or out of the solution, regardless of which action may later be selected in them.

Let $\mathcal{I}$ denote the set of planning units and let $w_i \in \{0, 1\}$ denote the binary variable indicating whether planning unit $i \in \mathcal{I}$ is selected by the model.

This function defines two subsets:

- $\mathcal{I}^{in} \subseteq \mathcal{I}$, the planning units that must be included,
- $\mathcal{I}^{out} \subseteq \mathcal{I}$, the planning units that must be excluded.

Conceptually, these sets correspond to the following conditions:

- if $i \in \mathcal{I}^{in}$, then $w_i = 1$,

- if $i \in \mathcal{I}^{out}$, then $w_i = 0$.

These constraints are not imposed immediately by this function; instead, they are stored in the planning-unit table and enforced later when building the optimization model.

Philosophy

The role of `create_problem` is to construct and normalize the basic inputs of the planning problem. Locking planning units is treated as a separate modelling step so that users can define or revise selection restrictions after the `Problem` object has already been created.

In contrast, `add_constraint_locked_actions` is used to fix specific feasible `(pu, action)` decisions rather than whole planning units.

Supported input formats

For both `locked_in` and `locked_out`, the function accepts:

- `NULL`, meaning that no planning units are locked on that side,
- a single character string, interpreted as a column name in the raw planning-unit data,
- a logical vector of length `nrow(pu)`,
- a vector of planning-unit ids.

When a column name is supplied, the referenced column is coerced to logical. Numeric values are interpreted as non-zero = `TRUE`; character and factor values are interpreted using common logical strings such as `"true"`, `"t"`, `"1"`, `"yes"`, and `"y"`. Missing values are treated as `FALSE`.

Repeated calls

Calls accumulate compatible locks and preserve omitted arguments. Repeating the same lock is idempotent. A planning unit cannot be locked in after it was locked out, or vice versa. Conflicts with existing action locks are also rejected. To change a lock, rebuild from the problem before it was added.

Consistency checks

The function checks that no planning unit is simultaneously assigned to both `locked_in` and `locked_out`. If such conflicts are found, an error is raised.

Value

An updated `Problem` object in which the planning-unit table contains logical columns `locked_in` and `locked_out`.

See Also

`create_problem`, `add_actions`, `add_constraint_locked_actions`, `add_constraint_locked_pu`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

pu <- example_data$planning_units
pu$lock_col <- pu$id %in% c(1, 4)
pu$out_col <- pu$id == 5
```

```
p <- create_problem(
  pu = pu,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
)

# 1) Lock by planning-unit ids
p1 <- add_constraint_locked_planning_units(
  x = p,
  locked_in = c(1, 3),
  locked_out = c(5)
)

p1$data$pu[, c("id", "locked_in", "locked_out")]

# 2) Read lock information from raw planning-unit data columns
p2 <- add_constraint_locked_planning_units(
  x = p,
  locked_in = "lock_col",
  locked_out = "out_col"
)

p2$data$pu[, c("id", "locked_in", "locked_out")]

# 3) Use logical vectors
p3 <- add_constraint_locked_planning_units(
  x = p,
  locked_in = pu$id %in% c(1, 3),
  locked_out = pu$id == 4
)

p3$data$pu[, c("id", "locked_in", "locked_out")]
```

`add_constraint_locked_pu` *Add locked planning units to a problem*

Description

[Deprecated]

`add_constraint_locked_pu()` has been replaced by `add_constraint_locked_planning_units`.

Usage

```
add_constraint_locked_pu(x, locked_in = NULL, locked_out = NULL)
```

Arguments

x	A Problem object created with create_problem.
locked_in	Optional locked-in specification. It may be NULL, a column name in the raw planning-unit data, a logical vector, or a vector of planning-unit ids.
locked_out	Optional locked-out specification. It may be NULL, a column name in the raw planning-unit data, a logical vector, or a vector of planning-unit ids.

Value

An updated Problem object in which the planning-unit table contains logical columns locked_in and locked_out.

See Also

add_constraint_locked_planning_units

---

add_constraint_targets_absolute

Add absolute targets

Description

Add feature-level absolute targets to a planning problem.

These targets are stored in the problem object and later translated into linear constraints when the optimization model is built. Absolute targets are interpreted directly in the same units as the feature contributions used by the model. Each call appends one or more target definitions to the problem. This makes it possible to combine multiple target rules, including targets associated with different action subsets.

Usage

```
add_constraint_targets_absolute(
  x,
  targets,
  features = NULL,
  actions = NULL,
  label = NULL
)
```

Arguments

x	A Problem object.
targets	Target specification. This is interpreted as an absolute target value in the same units as the modelled feature contributions. It may be a scalar, vector, named vector, or data.frame. See Details.

`features` Optional feature specification indicating which features the supplied target values refer to when `targets` does not identify features explicitly. If `NULL`, all features are targeted.

`actions` Optional character vector indicating which actions count toward target achievement. Entries may match action ids, `action_set` labels, or both. If `NULL`, all actions count.

`label` Optional character string stored with the targets for reporting and bookkeeping.

Details

Calls accumulate targets for distinct feature/action scopes. A second target for the same feature and action scope raises an error immediately, even if it uses relative units or a different label. To change a target, rebuild from the problem before it was added.

Use this function when target requirements are naturally expressed in the original units of the modelled feature contributions, rather than as proportions of reference-scenario totals.

Let $\mathcal{F}$ denote the set of features. For each targeted feature $f \in \mathcal{F}$, this function stores an absolute target threshold $T_f \geq 0$.

With at most one action per unit, each target is interpreted as a lower-bound constraint on post-action amounts of the form:

$$\sum_{(i,a)\in\mathcal{D}_f^\star} c_{iaf} x_{ia} \geq T_f,$$

where:

- $i \in \mathcal{I}$ indexes planning units,
- $a \in \mathcal{A}$ indexes actions,
- x_{ia} indicates whether action a is selected in planning unit i,
- c_{iaf} is the contribution of that action to feature f,
- $\mathcal{D}_f^\star$ is the subset of planning unit–action pairs allowed to count toward the target for feature f.

In the absolute case, the stored target threshold is simply:

$$T_f = t_f,$$

where t_f is the user-supplied target value for feature f.

With concurrent actions, the model instead combines signed changes within each unit and feature and counts the reference once:

$$\sum_i \left(r_{if} s_i^\star + \Delta_{if}^\star(x)\right) \geq T_f.$$

Here $s_i^\star$ indicates that at least one action in the target scope is selected. Joint corrections contribute only when every member belongs to that scope. Individual effects are additive when no interaction is supplied. The same aggregation is used to report target achievement.

The `actions` argument restricts which actions may contribute toward achievement of the target, but it does not modify the value of T_f itself.

The `targets` argument is parsed by `.pa_parse_targets()` and may be supplied in several equivalent forms, including:

- a single numeric value recycled to all selected features,
- a numeric vector aligned to `features`,
- a named numeric vector where names identify features,
- a `data.frame` with `feature` and `target` columns.

If `targets` does not explicitly identify features:

- if `features = NULL`, the target is applied to all features,
- if `features` is supplied, the target values are interpreted with respect to that feature set.

Repeated calls append new target rules rather than replacing previous ones. This allows cumulative target modelling, including multiple rules on the same feature with different contributing action subsets.

Value

An updated `Problem` object with absolute targets appended to the stored target table.

See Also

`add_constraint_targets_relative`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

p <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  )

# Same absolute target for all features
p1 <- add_constraint_targets_absolute(p, 3)
p1$data$targets

# Different targets by feature
p2 <- add_constraint_targets_absolute(
  p,
  c("1" = 4, "2" = 2)
)
p2$data$targets

# Restrict which actions count toward target achievement
p3 <- add_constraint_targets_absolute(
```

```
  p,
  2,
  actions = "protect"
)
p3$data$targets
```

`add_constraint_targets_relative` *Add relative targets*

Description

Add feature-level relative targets to a planning problem.

These targets are stored in the problem object and later translated into linear constraints when the optimization model is built. Relative targets are supplied as proportions in $[0, 1]$ and are converted internally into absolute thresholds using the current total amount of each feature in the landscape.

Each call appends one or more target definitions to the problem. This makes it possible to combine multiple target rules, including targets associated with different action subsets.

Usage

```
add_constraint_targets_relative(
  x,
  targets,
  features = NULL,
  actions = NULL,
  label = NULL
)
```

Arguments

`x`	A `Problem` object.
`targets`	Target specification as proportions in $[0, 1]$. It may be a scalar, vector, named vector, or `data.frame`. See Details.
`features`	Optional feature specification indicating which features the supplied target values refer to when `targets` does not identify features explicitly. If `NULL`, all features are targeted.
`actions`	Optional character vector indicating which actions count toward target achievement. Entries may match action ids, `action_set` labels, or both. If `NULL`, all actions count.
`label`	Optional character string stored with the targets for reporting and bookkeeping.

Details

Use this function when target requirements are naturally expressed as proportions of reference-scenario feature totals rather than in original feature units.

Let $\mathcal{F}$ denote the set of features. For each targeted feature $f \in \mathcal{F}$, let B_f denote the reference-scenario total amount of that feature in the landscape, as computed by `.pa_feature_totals()`.

If the user supplies a relative target $r_f \in [0, 1]$, then this function converts it to an absolute threshold:

$$T_f = r_f \times B_f.$$

The absolute threshold T_f is stored in `target_value`, while:

- the original user-supplied proportion r_f is stored in `target_raw`,
- the baseline total B_f is stored in `basis_total`.

When the optimization model is built, the resulting target is interpreted as:

$$\sum_{(i,a) \in \mathcal{D}_f^\star} c_{iaf} x_{ia} \geq T_f,$$

where:

- $i \in \mathcal{I}$ indexes planning units,
- $a \in \mathcal{A}$ indexes actions,
- x_{ia} indicates whether action a is selected in planning unit i,
- c_{iaf} is the contribution of that action to feature f,
- $\mathcal{D}_f^\star$ is the subset of planning unit–action pairs allowed to count toward the target for feature f.

With concurrent actions, the model instead combines signed changes within each unit and feature and counts the reference once:

$$\sum_i \left(r_{if} s_i^\star + \Delta_{if}^\star(x) \right) \geq T_f.$$

Here $s_i^\star$ indicates that at least one action in the target scope is selected. Joint corrections contribute only when every member belongs to that scope. Individual effects are additive when no interaction is supplied. The same aggregation is used to report target achievement.

The `actions` argument restricts which actions may contribute toward target achievement, but it does not affect the baseline amount B_f used to compute the threshold. In other words, relative targets are always scaled against the full landscape reference scenario.

Therefore, `actions` changes who may satisfy the target, but not how the threshold itself is scaled.

The `targets` argument is parsed by `.pa_parse_targets()` and may be supplied in several equivalent forms, including:

- a single numeric value recycled to all selected features,
- a numeric vector aligned to `features`,
- a named numeric vector where names identify features,
- a `data.frame` with `feature` and `target` columns.

If `targets` does not explicitly identify features:

- if `features` = `NULL`, the target is applied to all features,
- if `features` is supplied, the target values are interpreted with respect to that feature set.

Relative targets must lie in $[0, 1]$. A second target for the same feature and action scope raises an error, including when an absolute target already exists. Labels and threshold values do not change target identity.

Repeated calls append new target rules rather than replacing previous ones. This allows cumulative target modelling, including multiple rules on the same feature with different contributing action subsets.

Value

An updated `Problem` object with relative targets appended to the stored target table.

See Also

`add_constraint_targets_absolute`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

p <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  )

# Require 30% of the baseline total for all features
p1 <- add_constraint_targets_relative(p, 0.3)
p1$data$targets

# Require 20% for one selected feature
p2 <- add_constraint_targets_relative(
  p,
  0.2,
  features = 1
)
p2$data$targets

# Restrict which actions count toward target achievement
p3 <- add_constraint_targets_relative(
  p,
  0.2,
```

```
  actions = "protect"
)
p3$data$targets
```

`add_effects` *Add action effects to a planning problem*

Description

Specify the expected consequences of feasible management actions for feature amounts in planning units, relative to a reference scenario.

Usage

```
add_effects(
  x,
  effects = NULL,
  effect_type = c("delta", "after"),
  effect_aggregation = c("sum", "mean"),
  component = c("any", "benefit", "loss"),
  raster_aggregation = c("sum", "mean"),
  raster_type = c("effect", "outcome")
)
```

Arguments

`x`	A `Problem` created by `create_problem` with feasible actions already defined by `add_actions`.
`effects`	A data frame with `action`, `feature`, optional `pu`, and exactly one of `effect`, `outcome`, or `relative_change`; a named list of action rasters; or `NULL` to register no effects. Historical formats remain supported.
`effect_type`	Deprecated interpretation argument: `"delta"` for signed changes or `"after"` for expected amounts. Legacy multipliers retain their original interpretation; omit for modern tables.
`effect_aggregation`	Deprecated raster aggregation argument; use `raster_aggregation` instead.
`component`	Deprecated sign filter: `"any"` retains all effects, `"benefit"` keeps positive changes, and `"loss"` keeps negative changes. These filters do not encode ecological desirability.
`raster_aggregation`	Aggregate raster values using `"sum"` (default) or `"mean"` for each planning unit.
`raster_type`	Interpret raster values as signed changes (`"effect"`, default) or expected amounts (`"outcome"`). Explicitly provide this or `raster_aggregation` to select the modern raster interface.

Details

Reference amounts, outcomes, and effects

The feature distribution supplied to `create_problem()` defines the reference amount r_{if} of feature f in planning unit i. This reference may describe current conditions, existing management, or a future without intervention. Let y_{iaf} be the expected amount under action a. Its signed effect is $e_{iaf} = y_{iaf} - r_{if}$. These quantities describe what an action changes, not whether that change is desirable. For example, increasing habitat may be desirable, whereas decreasing fuel load may also be desirable. Objectives and constraints determine how consequences are valued in a planning problem.

Three alternative tabular inputs

Provide `action`, `feature`, optionally `pu`, and exactly one of the following numeric columns:

- `effect`: signed absolute change e_{iaf}.
- `outcome`: expected amount y_{iaf} under the action.
- `relative_change`: proportional change c_{iaf}, with $y_{iaf} = r_{if}(1 + c_{iaf})$ and $e_{iaf} = r_{if}c_{iaf}$.

For a reference amount of 100, `effect = 30`, `outcome = 130`, and `relative_change = 0.30` are equivalent. Likewise, `relative_change = -0.20` represents a 20 percent decrease. These input columns are alternatives, not values to supply together.

Actions must first be registered with `add_actions()`. With `pu`, effects can vary by location; without it, each action/feature specification is expanded over feasible planning-unit/action pairs. Locked-out pairs are excluded. A feature can be identified by name or numeric identifier. Values must be finite and non-missing; resulting outcomes must be non-negative. Reference amounts and outcomes must have compatible units and, when applicable, a common time horizon. Missing reference amounts are treated as zero: a relative change cannot create a positive outcome from a zero reference, so use `effect` or `outcome` in that case.

Stored representation

Regardless of the input format, `dist_effects` exposes `reference_amount`, `action_outcome`, and signed `effect`. For compatibility, it also retains `amount_after` (an alias of `action_outcome`) and columns named `benefit` and `loss`: these are strictly the positive and negative magnitudes of the signed change, $max(e, 0)$ and $max(-e, 0)$, not ecological judgements. A zero effect leaves the outcome equal to the reference amount.

Joint action effects

An `action` identifier may also refer to a set registered with `add_action_sets()`. Supply its *total* joint effect or outcome, not merely the interaction increment; individual and joint effects must be supplied in the same call. In the absence of a specified interaction, concurrent individual effects are treated as additive. Unspecified terms are assumed to be zero, which is a modelling assumption. Joint effects and signed interaction corrections are retained in separate internal tables; `dist_effects` represents individual actions. Cardinality constraints determine whether combinations may be selected.

Raster inputs

Supply a named list of `terra::SpatRaster` objects, one per action, each containing one layer per feature in feature-catalogue order. The problem must contain planning-unit polygons or a planning-unit raster. Use `raster_type = "effect"` for signed changes or `raster_type = "outcome"` for expected amounts, and choose `raster_aggregation = "sum"` or `"mean"` to aggregate values within

planning units. Ensure the aggregated raster amounts are comparable with the reference; relative-change rasters are not accepted directly.

Single definition and compatibility

Effects can be specified only once per problem. To compare effect scenarios, create separate problems from a common object before calling `add_effects()`. Historical `delta`, `after`, `multiplier`, `benefit`, and `loss` inputs and explicit `effect_type`, `effect_aggregation`, and `component` arguments remain available with deprecation warnings. Use the three modern tabular columns for new code.

Value

An updated `Problem` containing `dist_effects` and `effects_meta`; the stored effect table includes reference amounts, action outcomes, signed effects, and compatibility columns.

See Also

`add_actions`, `add_action_sets`

Examples

```
# EXAMPLE 1: Equivalent ways to describe action effects
#
# Consider two planning units and two features: habitat and fuel load.
# The reference amounts represent their values before implementing
# any management action.
#
# In planning unit 1, habitat = 100 and fuel load = 60.
# In planning unit 2, habitat = 40 and fuel load = 20.

p <- create_problem(
  pu = data.frame(id = 1:2, cost = 0),
  features = data.frame(id = 1:2, name = c("habitat", "fuel")),
  dist_features = data.frame(
    pu = c(1, 1, 2, 2), feature = c(1, 2, 1, 2),
    amount = c(100, 60, 40, 20)
  )
) |>
  add_actions(actions = data.frame(id = "restore"))

# Suppose restoration increases habitat in planning unit 1
# from 100 to 130. This response can be described in three
# equivalent ways:
#
#   effect          = +30   (absolute increase)
#   outcome         = 130   (amount after restoration)
#   relative_change = 0.30  (30% increase)
#
# Each call starts from the same base problem because effects
# can only be defined once per problem.

p_effect <- add_effects(p, data.frame(
```

```
  pu = 1, action = "restore", feature = "habitat", effect = 30
))

p_outcome <- add_effects(p, data.frame(
  pu = 1, action = "restore", feature = "habitat", outcome = 130
))

p_relative <- add_effects(p, data.frame(
  pu = 1, action = "restore", feature = "habitat",
  relative_change = 0.30
))

# Regardless of the input specification, multiscape stores
# the reference amount, signed effect, and action outcome.

p_effect$data$dist_effects[, c(
  "reference_amount", "effect", "action_outcome"
)]


# EXAMPLE 2: Effects that apply across planning units
#
# The 'pu' column is optional. When omitted, the specified
# change is applied to every feasible planning unit for
# the corresponding action.
#
# Here, restoration increases habitat by 25% in both units.
# Because their reference amounts differ (100 and 40),
# the resulting absolute effects also differ (+25 and +10).

p_global <- add_effects(p, data.frame(
  action = "restore", feature = "habitat",
  relative_change = 0.25
))

# Inspect how the same proportional change produces
# different absolute effects across planning units.

p_global$data$dist_effects[, c(
  "pu", "reference_amount", "effect"
)]


# EXAMPLE 3: Positive and negative effects
#
# A management action may simultaneously increase some
# features and decrease others.
#
# Suppose restoration increases habitat from 100 to 130
# but reduces fuel load from 60 to 45 in planning unit 1.
#
# Both responses are represented as signed effects:
#
```

```
#   habitat: +30
#   fuel:    -15
#
# Importantly, the sign describes the direction of change,
# not whether that change is desirable. Increasing habitat
# and reducing fuel load may both support management goals.

p_mixed <- add_effects(p, data.frame(
  pu = 1, action = "restore",
  feature = c("habitat", "fuel"),
  effect = c(30, -15)
))

# Inspect the direction and magnitude of each response.

p_mixed$data$dist_effects[, c(
  "feature_name", "effect", "action_outcome"
)]


# EXAMPLE 4: Spatial variation in action outcomes
#
# Use the bundled 64-unit landscape to illustrate how
# reference amounts and action outcomes vary spatially.
# No optimisation or solver is required.
#
# The example applies a hypothetical 25% increase in the
# first feature under the first available action.

if (requireNamespace("sf", quietly = TRUE)) {

  # Load the spatial planning units, features, reference
  # distributions, and available management actions.

  sim <- load_sim_multiaction()

  spatial_problem <- create_problem(
    pu = sim$planning_units,
    features = sim$features,
    dist_features = sim$dist_features,
    cost = "cost"
  ) |>
    add_actions(sim$actions, cost = sim$action_costs)

  # Select one feature and one management action.

  f <- sim$features$name[1]
  a <- sim$actions$id[1]

  # Define a hypothetical 25% increase relative to the
  # reference amount in every feasible planning unit.
  # Because reference amounts vary spatially, so do the
  # resulting absolute effects and action outcomes.
```

```
  spatial_problem <- add_effects(
    spatial_problem,
    data.frame(
      action = a,
      feature = f,
      relative_change = 0.25
    )
  )

  # Extract the reference and expected outcome for the
  # selected feature-action combination.

  vals <- subset(
    spatial_problem$data$dist_effects,
    action == a & feature_name == f,
    select = c("pu", "reference_amount", "action_outcome")
  )

  # Attach these values to the planning-unit geometries
  # so they can be displayed as spatial maps.

  mapped <- merge(
    sim$planning_units, vals,
    by.x = "id", by.y = "pu", all.x = TRUE
  )

  # Compare reference amounts with potential outcomes.
  # These maps represent the consequences of implementing
  # the action, not an optimised selection of actions.

  plot(mapped[c("reference_amount", "action_outcome")])
}


# EXAMPLE 5: Joint effects of multiple actions
#
# When multiple actions can be implemented together,
# their combined effect need not equal the sum of their
# individual effects.
#
# Consider two actions, 'restore' and 'control'.
# Their individual effects on habitat are +30 and +20.
# However, implementing both together produces a total
# effect of +70 rather than +50.
#
# This implies an additional interaction of +20:
#
#   restore effect          = +30
#   control effect          = +20
#   additional interaction  = +20
#   joint total effect      = +70
#
```

```
# The joint effect is supplied as the TOTAL change,
# not merely the additional interaction.

joint_base <- create_problem(
  data.frame(id = 10L, cost = 0),
  data.frame(id = 1L, name = "habitat"),
  data.frame(pu = 10L, feature = 1L, amount = 100)
) |>
  add_actions(
    data.frame(id = c("restore", "control")),
    cost = 1
  ) |>
  add_action_sets(
    list(restore_control = c("restore", "control"))
  ) |>
  add_constraint_action_cardinality(2, "max")

# Supply individual and joint effects in the same call.
# The combined action changes habitat from 100 to 170.

joint <- add_effects(joint_base, data.frame(
  action = c("restore", "control", "restore_control"),
  feature = "habitat",
  effect = c(30, 20, 70)
))

# Inspect the decomposition into individual effects and
# the additional correction associated with joint selection.
# The joint row stores a correction of +20, ensuring that
# the combined total is +70 rather than +90.

joint$data$effect_terms[, c(
  "action", "total_effect", "effect"
)]
```

`add_losses` *Add losses*

Description

Legacy convenience wrapper around `add_effects` that retains only negative signed changes (`effect < 0`). Negative change does not necessarily imply an ecological loss. For new code, prefer `add_effects()` to retain signed effects of both directions. Effects can be defined only once per problem.

Usage

```
add_losses(
  x,
  losses = NULL,
  effect_type = c("delta", "after"),
```

```
  effect_aggregation = c("sum", "mean")
)
```

Arguments

x	A Problem created by create_problem with feasible actions already defined by add_actions.
losses	Alias of effects, used for symmetry with add_benefits().
effect_type	Deprecated interpretation argument: "delta" for signed changes or "after" for expected amounts. Legacy multipliers retain their original interpretation; omit for modern tables.
effect_aggregation	Deprecated raster aggregation argument; use raster_aggregation instead.

Value

An updated Problem object containing:

dist_effects The canonical filtered effects table, containing only rows with loss > 0.

dist_loss A convenience table containing only the loss component.

losses_meta Metadata for the stored loss table.

See Also

add_effects, add_benefits, add_objective_min_loss

---

add_objective_max_benefit

Deprecated benefit objective

Description

Use add_objective_max_effect() for signed net effects.

Usage

```
add_objective_max_benefit(x, actions = NULL, features = NULL, alias = NULL)
```

Arguments

x	A Problem object.
actions	Optional subset of actions to include in the objective. Values may match x$data$actions$id and, if present, x$data$actions$action_set. If NULL, all actions are included.
features	Optional subset of features to include in the objective. Values may match x$data$features$id and, if present, x$data$features$name. If NULL, all features are included.
alias	Optional identifier used to register this objective for multi-objective workflows.

Value

An updated Problem object.

`add_objective_max_effect`

Add objective: maximize signed effect

Description

Maximize the total signed change generated by selected management actions relative to the reference scenario. Positive changes increase the objective; negative changes reduce it.

Usage

```
add_objective_max_effect(x, actions = NULL, features = NULL, alias = NULL)
```

Arguments

`x`	A `Problem` object.
`actions`	Optional subset of actions to include in the objective. Values may match `x$data$actions$id` and, if present, `x$data$actions$action_set`. If `NULL`, all actions are included.
`features`	Optional subset of features to include in the objective. Values may match `x$data$features$id` and, if present, `x$data$features$name`. If `NULL`, all features are included.
`alias`	Optional identifier used to register this objective for multi-objective workflows.

Details

Let $\Delta_{if}(x)$ be the signed change for planning unit i and feature f, after combining selected individual actions and their interaction corrections. The objective is

$$\max \sum_{i,f} \Delta_{if}(x).$$

For one action per unit this reduces to the sum of selected signed effects. With concurrent actions, supplied joint totals and inferred combinations are evaluated together; a negative interaction correction is not discarded.

The reference is the scenario supplied through `dist_features`, at the same horizon and in the same units as action outcomes. An unselected unit contributes zero change. Costs are not subtracted by this objective. A positive effect means an increase in the feature; select features for which an increase is desirable. Standardize incompatible units before aggregation, or register separate feature objectives using distinct aliases.

With `actions`, include only individual terms in that subset and joint corrections whose every member belongs to it. This is an accounting scope, not attribution of a cross-subset interaction to its individual members. With `features`, include only the selected features.

Value

An updated `Problem` object.

Changed behavior

This development version maximizes signed net change. Earlier versions maximized positive effects only. Models with negative effects or negative interaction corrections may therefore produce different solutions. The historical add_objective_max_benefit() name is deprecated in favor of add_objective_max_effect(). `get_features()` still reports positive gains separately in `selected_benefit`; the signed objective corresponds to `selected_net` when their scopes match.

Repeated calls

With `alias = NULL`, one explicit single objective can be defined per problem; a second unaliased definition raises an error. With an alias, objectives accumulate under distinct names; a repeated alias raises an error. Aliased definitions preserve an explicitly configured single objective. To compare alternatives, start from the problem before its objective was added.

References

Tallis, H., et al. (2021). Prioritizing actions: spatial action maps for conservation. *Annals of the New York Academy of Sciences*, 1505, 118–141. doi:10.1111/nyas.14651

Salgado-Rojas, J., Álvarez-Miranda, E., Hermoso, V., Garcia-Gonzalo, J., and Weintraub, A. (2020). A mixed integer programming approach for multi-action planning for threat management. *Ecological Modelling*, 418, 108901. doi:10.1016/j.ecolmodel.2019.108901

See Also

`add_objective_min_effect`, `add_effects`

Examples

```
# EXAMPLE: Maximise woodland gain in a fixed number of units
#
# The bundled landscape has spatially varying woodland reference
# amounts. Assume restoration increases these amounts by 50%.
# The absolute signed effect is therefore larger where the
# reference amount is greater.
sim <- load_sim_multiaction()

p <- create_problem(
  pu = sim$planning_units,
  features = sim$features,
  dist_features = sim$dist_features,
  cost = "cost"
) |>
  add_actions(data.frame(id = "restore"), cost = 1) |>
  add_effects(data.frame(
    action = "restore", feature = "woodland", relative_change = 0.50
  )) |>
```

```
  add_constraint_budget(12, "equal", include_pu_cost = FALSE) |>
  add_objective_max_effect(features = "woodland", alias = "woodland_gain")

# Every restoration costs one monetary unit, so exactly 12 units
# must be managed. The objective favours the largest woodland gains.
if (requireNamespace("rcbc", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  solutions <- solve(set_solver_cbc(p, time_limit = 30, verbose = FALSE))
  get_objectives(solutions, format = "wide")
  plot_spatial_actions(solutions, layout = "single")
}
```

`add_objective_max_net_profit` *Add objective: maximize net profit*

Description

Define an objective that maximizes net profit by combining profits with optional planning-unit and action-cost penalties.

Usage

```
add_objective_max_net_profit(
  x,
  profit_col = "profit",
  include_pu_cost = TRUE,
  include_action_cost = TRUE,
  actions = NULL,
  alias = NULL
)
```

Arguments

`x`	A `Problem` object.
`profit_col`	Character string giving the profit column in the stored profit table.
`include_pu_cost`	Logical. If `TRUE`, subtract planning-unit costs.
`include_action_cost`	Logical. If `TRUE`, subtract action costs.
`actions`	Optional subset of actions to include in the profit and action-cost terms. Values may match `x$data$actions$id` and, if present, `x$data$actions$action_set`.
`alias`	Optional identifier used to register this objective for multi-objective workflows.

Details

Use this function when decisions generate returns and the objective should optimize the resulting net balance after subtracting selected cost components.

Let:

- $x_{ia} \in \{0, 1\}$ denote whether action a is selected in planning unit i,
- $w_i \in \{0, 1\}$ denote whether planning unit i is selected,
- π_{ia} denote the profit associated with decision (i, a),
- $c_i^{PU} \geq 0$ denote the planning-unit cost,
- $c_{ia}^{A} \geq 0$ denote the action cost.

In its most general form, the objective is:

$$\max \left(\sum_{(i,a) \in \mathcal{D}^\star} \pi_{ia} x_{ia} - \sum_{i \in \mathcal{I}} c_i^{PU} w_i - \sum_{(i,a) \in \mathcal{D}^\star} c_{ia}^{A} x_{ia} \right),$$

where $\mathcal{D}^\star$ denotes the subset of feasible planning unit–action decisions included in the objective.

If `actions = NULL`, all feasible actions contribute to both the profit term and the action-cost term.

If `actions` is provided, the profit term and the action-cost term are restricted to that subset. The planning-unit cost term, if included, remains global.

If `include_pu_cost = FALSE`, the planning-unit cost term is omitted.

If `include_action_cost = FALSE`, the action-cost term is omitted.

Value

An updated `Problem` object.

Repeated calls

With `alias = NULL`, one explicit single objective can be defined per problem; a second unaliased definition raises an error. With an alias, objectives accumulate under distinct names; a repeated alias raises an error. Aliased definitions preserve an explicitly configured single objective. To compare alternatives, start from the problem before its objective was added.

References

Polasky, S., et al. (2008). Where to put things? Spatial land management to sustain biodiversity and economic returns. *Biological Conservation*, 141, 1505–1524. doi:10.1016/j.biocon.2008.03.022

Chan, K. M. A., Hoshizaki, L., and Klinkenberg, B. (2011). Ecosystem services in conservation planning: targeted benefits vs. co-benefits or costs? *PLOS ONE*, 6, e24378. doi:10.1371/journal.pone.0024378

See Also

`add_objective_max_profit`, `add_objective_min_cost`

Examples

```
# EXAMPLE: Maximise returns minus implementation costs
#
# Use the same west-to-east profit pattern as above, but now
# charge 9.5 monetary units for every selected action. Some
# locations may no longer generate a positive net return.
sim <- load_sim_multiaction()
returns <- sim$action_costs[, c("pu", "action")]
x_coord <- sim$planning_units$x[
  match(returns$pu, sim$planning_units$id)
]
returns$profit <- ifelse(
  returns$action == "protect", 12 - x_coord, 4 + x_coord
)

p <- create_problem(
  pu = sim$planning_units,
  features = sim$features,
  dist_features = sim$dist_features,
  cost = "cost"
) |>
  add_actions(sim$actions, cost = 9.5) |>
  add_profit(returns) |>
  add_objective_max_net_profit(include_pu_cost = FALSE, alias = "net_profit")

# Unlike gross profit maximisation, leaving a unit unmanaged
# can now be optimal when its best return is below its cost.
if (requireNamespace("rcbc", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  solutions <- solve(set_solver_cbc(p, time_limit = 30, verbose = FALSE))
  get_objectives(solutions, format = "wide")
  plot_spatial_actions(solutions, layout = "single")
}
```

---

add_objective_max_profit

Add objective: maximize profit

Description

Define an objective that maximizes total profit from selected planning unit–action decisions.

Usage

```
add_objective_max_profit(
  x,
  profit_col = "profit",
  actions = NULL,
```

```
  alias = NULL
)
```

Arguments

x	A Problem object.
profit_col	Character string giving the profit column in the stored profit table.
actions	Optional subset of actions to include. Values may match x$data$actions$id and, if present, x$data$actions$action_set. If NULL, all actions are included.
alias	Optional identifier used to register this objective for multi-objective workflows.

Details

Use this function when the objective is to maximize gross economic return, without subtracting planning-unit or action costs.

Let $x_{ia} \in \{0, 1\}$ denote whether action a is selected in planning unit i, and let π_{ia} denote the profit associated with that decision, as taken from column profit_col in the stored profit table.

If all actions are included, the objective is:

$$\max \sum_{(i,a)\in\mathcal{D}} \pi_{ia} x_{ia},$$

where $\mathcal{D}$ denotes the set of feasible planning unit–action decisions.

If actions is provided, only the selected subset contributes to the objective. Letting $\mathcal{D}^\star$ denote the feasible decisions whose action belongs to the selected subset, the objective becomes:

$$\max \sum_{(i,a)\in\mathcal{D}^\star} \pi_{ia} x_{ia}.$$

This objective considers profit only. It does not subtract planning-unit costs or action costs. For a net-profit formulation, use add_objective_max_net_profit.

Value

An updated Problem object.

Repeated calls

With alias = NULL, one explicit single objective can be defined per problem; a second unaliased definition raises an error. With an alias, objectives accumulate under distinct names; a repeated alias raises an error. Aliased definitions preserve an explicitly configured single objective. To compare alternatives, start from the problem before its objective was added.

References

Polasky, S., et al. (2008). Where to put things? Spatial land management to sustain biodiversity and economic returns. *Biological Conservation*, 141, 1505–1524. doi:10.1016/j.biocon.2008.03.022

See Also

add_objective_min_cost, add_objective_max_net_profit

Examples

```
# EXAMPLE: Maximise gross economic return across the landscape
#
# Assign hypothetical profits that favour protection in the west
# and restoration in the east. All values are positive.
sim <- load_sim_multiaction()
returns <- sim$action_costs[, c("pu", "action")]
x_coord <- sim$planning_units$x[
  match(returns$pu, sim$planning_units$id)
]
returns$profit <- ifelse(
  returns$action == "protect", 12 - x_coord, 4 + x_coord
)

p <- create_problem(
  pu = sim$planning_units,
  features = sim$features,
  dist_features = sim$dist_features,
  cost = "cost"
) |>
  add_actions(sim$actions, cost = 9.5) |>
  add_profit(returns) |>
  add_objective_max_profit(alias = "profit")

# Implementation costs are NOT deducted in this objective.
# With positive returns and at most one action per unit, the
# model chooses the more profitable action in each unit.
if (requireNamespace("rcbc", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  solutions <- solve(set_solver_cbc(p, time_limit = 30, verbose = FALSE))
  get_objectives(solutions, format = "wide")
  plot_spatial_actions(solutions, layout = "single")
}
```

---

add_objective_min_cost

Add objective: minimize cost

Description

Define an objective that minimizes the total cost of the solution.

Depending on the function arguments, the objective may include planning-unit costs, action costs, or both. Action costs can optionally be restricted to a subset of actions.

Usage

```
add_objective_min_cost(
  x,
  include_pu_cost = TRUE,
  include_action_cost = TRUE,
  actions = NULL,
  alias = NULL
)
```

Arguments

x	A Problem object.
include_pu_cost	Logical. If TRUE, include planning-unit costs in the objective.
include_action_cost	Logical. If TRUE, include action costs in the objective.
actions	Optional subset of actions to include in the action-cost component. Values may match x$data$actions$id and, if present, x$data$actions$action_set. If NULL, all feasible actions are included in the action-cost term.
alias	Optional identifier used to register this objective for multi-objective workflows.

Details

Use this function when the planning problem is framed primarily as a cost-minimization problem, with costs arising from planning-unit selection, action implementation, or both.

Let $\mathcal{I}$ be the set of planning units and let $\mathcal{D} \subseteq \mathcal{I} \times \mathcal{A}$ denote the set of feasible planning unit–action decisions.

Let:

- $w_i \in \{0, 1\}$ denote whether planning unit i is selected,
- $x_{ia} \in \{0, 1\}$ denote whether action a is selected in planning unit i,
- $c_i^{PU} \geq 0$ denote the planning-unit cost of unit i,
- $c_{ia}^{A} \geq 0$ denote the cost of selecting action a in planning unit i.

The most general form of this objective is:

$$\min \left(\sum_{i \in \mathcal{I}} c_i^{PU} w_i + \sum_{(i,a) \in \mathcal{D}^\star} c_{ia}^{A} x_{ia} \right),$$

where $\mathcal{D}^\star$ denotes the subset of feasible decisions whose action contributes to the action-cost term.

If include_pu_cost = FALSE, the planning-unit cost term is omitted.

If include_action_cost = FALSE, the action-cost term is omitted.

If actions = NULL, all feasible actions contribute to the action-cost term. If actions is supplied, only the selected subset contributes to that term. Planning-unit costs are never subset by actions; they are always global whenever include_pu_cost = TRUE.

Value

An updated `Problem` object.

Repeated calls

With `alias = NULL`, one explicit single objective can be defined per problem; a second unaliased definition raises an error. With an alias, objectives accumulate under distinct names; a repeated alias raises an error. Aliased definitions preserve an explicitly configured single objective. To compare alternatives, start from the problem before its objective was added.

References

Ball, I. R., Possingham, H. P., and Watts, M. E. (2009). Marxan and relatives: software for spatial conservation prioritisation. In *Spatial Conservation Prioritization: Quantitative Methods and Computational Tools*, pp. 185–195. doi:10.1093/oso/9780199547760.003.0014

Salgado-Rojas, J., Hermoso, V., and Álvarez-Miranda, E. (2025). prioriactions: multi-action management planning in R. *Methods in Ecology and Evolution*, 16, 1028–1039. doi:10.1111/2041-210X.14220

See Also

`add_objective_max_profit`, `add_objective_max_net_profit`

Examples

```
# EXAMPLE: Minimum-cost management with a woodland target
#
# Use the 64 planning-unit landscape. The hypothetical outcome of 1
# credits one unit of woodland for either action in each selected unit.
# A target of 12 therefore requires at least 12 interventions.
sim <- load_sim_multiaction()

p <- create_problem(
  pu = sim$planning_units,
  features = sim$features,
  dist_features = sim$dist_features,
  cost = "cost"
) |>
  add_actions(sim$actions, cost = sim$action_costs) |>
  add_effects(data.frame(
    action = c("protect", "restore"),
    feature = "woodland", outcome = 1
  )) |>
  add_constraint_targets_absolute(12, features = "woodland") |>
  add_objective_min_cost(include_pu_cost = FALSE, alias = "cost")

# The solver selects the least costly set of actions that meets the target.
# This is a single-objective model: no set_method_*() call is needed.
if (requireNamespace("rcbc", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  solutions <- solve(set_solver_cbc(p, time_limit = 30, verbose = FALSE))
```

```
  get_objectives(solutions, format = "wide")
  plot_spatial_actions(solutions, layout = "single")
}
```

`add_objective_min_effect` *Add objective: minimise signed effect*

Description

Minimise total signed changes relative to the reference. Whether a decrease is desirable depends on the feature being optimised.

Usage

```
add_objective_min_effect(x, actions = NULL, features = NULL, alias = NULL)
```

Arguments

`x`	A `Problem` object.
`actions`	Optional subset of actions to include in the objective. Values may match `x$data$actions$id` and, if present, `x$data$actions$action_set`. If `NULL`, all actions are included.
`features`	Optional subset of features to include in the objective. Values may match `x$data$features$id` and, if present, `x$data$features$name`. If `NULL`, all features are included.
`alias`	Optional identifier used to register this objective for multi-objective workflows.

Details

For the selected actions and features, this objective minimises $\sum_{i,f} \Delta_{if}(x)$, where $\Delta_{if}(x)$ is the signed change after accounting for joint effects. Increases and decreases can offset one another across units or features.

For instance, minimising signed fuel-load change favours reductions in fuel load. Specify a feature subset when features are measured in incompatible units. Unselected units contribute zero change, and implementation costs are not subtracted.

This differs from the deprecated `add_objective_min_loss()`, which penalises only the negative part of aggregated changes.

Value

An updated Problem object.

References

Related spatial fuel-management applications (not necessarily using the identical signed-effect objective) include:

Minas, J. P., Hearne, J. W., and Martell, D. L. (2014). A spatial optimisation model for multi-period landscape level fuel management to mitigate wildfire impacts. *European Journal of Operational Research*, 232, 412–422. doi:10.1016/j.ejor.2013.07.026

Williams, B. A., Shoo, L. P., Wilson, K. A., and Beyer, H. L. (2017). Optimising the spatial planning of prescribed burns to achieve multiple objectives in a fire-dependent ecosystem. *Journal of Applied Ecology*. doi:10.1111/13652664.12920

See Also

add_objective_max_effect(), add_objective_min_loss()

Examples

```
# EXAMPLE: Minimise the signed change in fuel load
#
# Use the geometry from the bundled 64-unit landscape, with a simple
# hypothetical fuel-load feature that increases from west to east.
# A 25% reduction has a NEGATIVE signed effect, so minimising the
# effect favours the units with the greatest fuel-load reduction.
sim <- load_sim_multiaction()

p <- create_problem(
  pu = sim$planning_units,
  features = data.frame(id = 1L, name = "fuel_load"),
  dist_features = data.frame(
    pu = sim$planning_units$id, feature = 1L,
    amount = 10 + sim$planning_units$x
  ),
  cost = "cost"
) |>
  add_actions(data.frame(id = "thin"), cost = 1) |>
  add_effects(data.frame(
    action = "thin", feature = "fuel_load", relative_change = -0.25
  )) |>
  add_constraint_budget(12, "equal", include_pu_cost = FALSE) |>
  add_objective_min_effect(features = "fuel_load", alias = "fuel_reduction")

# With one monetary unit per action, the budget equality selects
# exactly 12 units. Without it, this negative-effect objective
# could favour implementing the action everywhere.
if (requireNamespace("rcbc", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  solutions <- solve(set_solver_cbc(p, time_limit = 30, verbose = FALSE))
  get_objectives(solutions, format = "wide")
  plot_spatial_actions(solutions, layout = "single")
}
```

`add_objective_min_fragmentation_action` *Add objective: minimise action fragmentation*

Description

Encourage spatial cohesion separately for each selected action, rather than only for the union of managed planning units.

Usage

```
add_objective_min_fragmentation_action(
  x,
  relation_name = "boundary",
  weight_multiplier = 1,
  action_weights = NULL,
  actions = NULL,
  alias = NULL
)
```

Arguments

`x`	A `Problem` object.
`relation_name`	Character string giving the name of the spatial relation to use. The relation must already exist in `x$data$spatial_relations`.
`weight_multiplier`	Numeric scalar greater than or equal to zero. Global multiplier applied to the relation weights when the objective is built.
`action_weights`	Optional action weights. Either a named numeric vector with names equal to action ids, or a `data.frame` with columns `action` and `weight`. These weights scale the contribution of each action to the final objective.
`actions`	Optional subset of actions to include. Values may match `x$data$actions$id` and, if present, `x$data$actions$action_set`. If `NULL`, all actions are included.
`alias`	Optional identifier used to register this objective for multi-objective workflows.

Details

This objective uses the action-selection decisions x_{ia} and a previously registered spatial relation. For neighbouring units i and j, $b_{ija} = x_{ia} \wedge x_{ja}$ represents whether the same action a occurs in both units. Adjacency of different actions does not form a continuous patch of either action.

For each action, the relation is restricted automatically to its feasible planning-unit/action pairs (as supplied through `include_pairs` and retained in the compiled model). An off-diagonal link contributes only if that action is feasible at both endpoints. A feasible but unselected action remains in this domain. No extra argument or correction objective is needed. Self-boundary weights contribute only where that action is feasible.

Use `actions` to select which action patterns contribute, `action_weights` to adjust their relative importance, and `weight_multiplier` to scale spatial relation weights.

Unlike `add_objective_min_fragmentation_planning_units()`, this objective distinguishes action identities. Combine it with coverage or allocation requirements so an empty plan is not optimal.

Value

An updated `Problem` object.

Repeated calls

With `alias = NULL`, one explicit single objective can be defined per problem; a second unaliased definition raises an error. With an alias, objectives accumulate under distinct names; a repeated alias raises an error. Aliased definitions preserve an explicitly configured single objective. To compare alternatives, start from the problem before its objective was added.

References

Salgado-Rojas, J., Álvarez-Miranda, E., Hermoso, V., Garcia-Gonzalo, J., and Weintraub, A. (2020). A mixed integer programming approach for multi-action planning for threat management. *Ecological Modelling*, 418, 108901. doi:10.1016/j.ecolmodel.2019.108901

Álvarez-Miranda, E., Salgado-Rojas, J., Hermoso, V., Garcia-Gonzalo, J., and Weintraub, A. (2020). An integer programming method for the design of multi-criteria multi-action conservation plans. *Omega*, 92, 102147. doi:10.1016/j.omega.2019.102147

See Also

`add_objective_min_fragmentation_planning_units`, `add_spatial_boundary`, `add_spatial_relations`

Examples

```
# EXAMPLE: Form separate cohesive patches for two actions
#
# Each action costs one unit. Two action-specific budget equalities
# require exactly four protection and four restoration units.
# The default one-action-per-unit rule prevents overlapping actions.
sim <- load_sim_multiaction()
# Use a 4 by 4 subset so this example solves quickly.
planning_units <- subset(sim$planning_units, x < 4 & y < 4)
dist_features <- subset(sim$dist_features, pu %in% planning_units$id)

p <- create_problem(
  pu = planning_units,
  features = sim$features,
  dist_features = dist_features,
  cost = "cost"
) |>
  add_actions(sim$actions, cost = 1) |>
  add_spatial_boundary(name = "boundary", include_self = TRUE) |>
  add_constraint_budget(
    4, "equal", actions = "protect", include_pu_cost = FALSE
```

```
  ) |>
  add_constraint_budget(
    4, "equal", actions = "restore", include_pu_cost = FALSE
  ) |>
  add_objective_min_fragmentation_action(relation_name = "boundary", alias = "action_fragmentation")

# Unlike planning-unit fragmentation, this objective measures
# the cohesion of each action's selected units separately.
if (requireNamespace("rcbc", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  solutions <- solve(set_solver_cbc(p, gap_limit = 0, time_limit = 5, verbose = FALSE))
  get_objectives(solutions, format = "wide")
  plot_spatial_actions(solutions, layout = "single")
}
```

`add_objective_min_fragmentation_planning_units` *Add objective: minimise planning-unit fragmentation*

Description

Encourage cohesion of the selected planning units, regardless of which action is implemented within each unit.

Usage

```
add_objective_min_fragmentation_planning_units(
  x,
  relation_name = "boundary",
  weight_multiplier = 1,
  alias = NULL
)
```

Arguments

`x`	A `Problem` object.
`relation_name`	Character string giving the name of the spatial relation to use. The relation must already exist in `x$data$spatial_relations`.
`weight_multiplier`	Numeric scalar greater than or equal to zero. Global multiplier applied to the relation weights when the objective is built.
`alias`	Optional identifier used to register this objective for multi-objective workflows.

Details

This objective uses planning-unit selection variables w_i and a spatial relation registered with `add_spatial_relations()` or `add_spatial_boundary()`. Relation weights ω_{ij} are scaled by `weight_multiplier`.

The underlying model uses $y_{ij} = w_i \wedge w_j$ to record whether neighbouring units are both selected, encouraging spatially consolidated selections. Action identities do not enter this spatial criterion: adjacent units receiving different actions are part of the same selected planning-unit set.

Without another requirement, selecting no units can be optimal. Combine this objective with a budget equality, an area requirement, or an ecological target that ensures meaningful management activity. For action-specific cohesion, see `add_objective_min_fragmentation_action()`.

Value

An updated `Problem` object.

Repeated calls

With `alias = NULL`, one explicit single objective can be defined per problem; a second unaliased definition raises an error. With an alias, objectives accumulate under distinct names; a repeated alias raises an error. Aliased definitions preserve an explicitly configured single objective. To compare alternatives, start from the problem before its objective was added.

References

Ball, I. R., Possingham, H. P., and Watts, M. E. (2009). Marxan and relatives: software for spatial conservation prioritisation. In *Spatial Conservation Prioritization: Quantitative Methods and Computational Tools*, pp. 185–195. doi:10.1093/oso/9780199547760.003.0014

Önal, H., and Briers, R. A. (2002). Incorporating spatial criteria in optimum reserve network selection. *Proceedings of the Royal Society B*, 269, 2437–2441. doi:10.1098/rspb.2002.2183

See Also

`add_spatial_boundary`, `add_spatial_relations`, `add_objective_min_fragmentation_action`, `add_objective_min_fragmentation_pu`

Examples

```
# EXAMPLE: Select a cohesive set of 12 planning units
#
# With one feasible action costing one unit, a budget equality
# forces exactly 12 selected units. Without this requirement,
# minimising fragmentation alone could select no units.
sim <- load_sim_multiaction()

p <- create_problem(
  pu = sim$planning_units,
  features = sim$features,
  dist_features = sim$dist_features,
  cost = "cost"
) |>
```

```
  add_actions(data.frame(id = "restore"), cost = 1) |>
  add_spatial_boundary(name = "boundary", include_self = TRUE) |>
  add_constraint_budget(12, "equal", include_pu_cost = FALSE) |>
  add_objective_min_fragmentation_planning_units(
    relation_name = "boundary", alias = "pu_fragmentation"
  )

# Cohesion is evaluated for the selected planning-unit pattern,
# irrespective of action identity. No additional objective is used.
if (requireNamespace("rcbc", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  solutions <- solve(set_solver_cbc(p, time_limit = 30, verbose = FALSE))
  get_objectives(solutions, format = "wide")
  plot_spatial_planning_units(solutions)
}
```

`add_objective_min_fragmentation_pu` *Add objective: minimize planning-unit fragmentation*

Description

[Deprecated]

`add_objective_min_fragmentation_pu()` has been replaced by `add_objective_min_fragmentation_planning_units`

Usage

```
add_objective_min_fragmentation_pu(
  x,
  relation_name = "boundary",
  weight_multiplier = 1,
  alias = NULL
)
```

Arguments

`x`	A `Problem` object.
`relation_name`	Character string giving the name of the spatial relation to use. The relation must already exist in `x$data$spatial_relations`.
`weight_multiplier`	Numeric scalar greater than or equal to zero. Global multiplier applied to the relation weights when the objective is built.
`alias`	Optional identifier used to register this objective for multi-objective workflows.

Value

An updated `Problem` object.

Repeated calls

With `alias = NULL`, one explicit single objective can be defined per problem; a second unaliased definition raises an error. With an alias, objectives accumulate under distinct names; a repeated alias raises an error. Aliased definitions preserve an explicitly configured single objective. To compare alternatives, start from the problem before its objective was added.

See Also

`add_objective_min_fragmentation_planning_units`

---

add_objective_min_intervention_impact

Add objective: minimize intervention impact

Description

[Deprecated]

Deprecated; retained temporarily with its original behavior.

Define an objective that minimizes the impact associated with selecting planning units for intervention.

This objective uses planning-unit selection variables rather than summing the same impact repeatedly over multiple actions. As a result, each planning unit contributes at most once to the objective, regardless of how many feasible actions exist in that unit.

Usage

```
add_objective_min_intervention_impact(
  x,
  impact_col = "amount",
  features = NULL,
  actions = NULL,
  alias = NULL
)
```

Arguments

`x`	A `Problem` object.
`impact_col`	Character string giving the column in the feature-distribution table that contains the per-`(pu, feature)` impact amount. The default is `"amount"`.
`features`	Optional subset of features to include. Values may match `x$data$features$id` and, if present, `x$data$features$name`.
`actions`	Optional subset of actions used to define the intervention context. Values may match `x$data$actions$id` and, if present, `x$data$actions$action_set`.
`alias`	Optional identifier used to register this objective for multi-objective workflows.

Details

The legacy objective minimizes the reference amount in units where at least one action in the selected scope is executed, counting each unit once. Its original arguments, objective sense, and single/MO formulations remain available during the transition.

Value

An updated `Problem` object.

Migration

This function will be removed in a future version of multiscape. New workflows should express action consequences with `add_effects` and optimize signed changes with `add_objective_max_effect`.

This is not an automatic replacement. For deficit-based prioritization, let q_{if} be the reference amount and M_f a common ceiling for feature f. Supply $M_f - q_{if}$ as the restoration effect, or M_f as its outcome. If every plan restores exactly K units and at most one scoped action is selected in each unit, then

$$\sum_i (M_f - q_{if}) r_i = K M_f - \sum_i q_{if} r_i.$$

Maximizing this deficit is equivalent to minimizing the old baseline sum. Equal area fixes K only when units have equal effective action areas. With variable effort, varying ceilings, or concurrent scoped actions, equivalence is not guaranteed. The common ceiling is a modelling assumption, not an empirical estimate of restoration response.

Repeated calls

With `alias = NULL`, one explicit single objective can be defined per problem; a second unaliased definition raises an error. With an alias, objectives accumulate under distinct names; a repeated alias raises an error. Aliased definitions preserve an explicitly configured single objective. To compare alternatives, start from the problem before its objective was added.

See Also

`add_objective_max_effect`, `add_objective_min_loss`

Examples

```
# Equal-area units with a fixed restoration effort of two units.
p <- create_problem(
  data.frame(id = 1:3, cost = 1, area = 1),
  data.frame(id = 1, name = "service"),
  data.frame(pu = 1:3, feature = 1, amount = c(0.2, 0.6, 0.9))
) |>
  add_actions(data.frame(id = "restore"), cost = 1) |>
  add_effects(data.frame(action = "restore", feature = "service", outcome = 1)) |>
  add_constraint_area(2, "equal", tolerance = 0, actions = "restore") |>
  add_objective_max_effect(features = "service", actions = "restore")
p$data$model_args
```

`add_objective_min_loss` *Deprecated objective: minimize loss*

Description

Minimize deterioration relative to the reference scenario. Positive changes in other planning units or features do not offset these losses.

Usage

```
add_objective_min_loss(x, actions = NULL, features = NULL, alias = NULL)
```

Arguments

`x`	A `Problem` object.
`actions`	Optional subset of actions to include in the objective. Values may match `x$data$actions$id` and, if present, `x$data$actions$action_set`. If `NULL`, all actions are included.
`features`	Optional subset of features to include in the objective. Values may match `x$data$features$id` and, if present, `x$data$features$name`. If `NULL`, all features are included.
`alias`	Optional identifier used to register this objective for multi-objective workflows.

Details

Let $\Delta_{if}(x)$ be the signed change for planning unit i and feature f, after combining selected individual actions and their interaction corrections. The objective is

$$\min \sum_{i,f} \max(-\Delta_{if}(x), 0).$$

The positive/negative split is applied after joint aggregation within each unit and feature, before summing across units or features. A negative interaction correction does not itself represent a loss if the final change remains positive. Conversely, gains elsewhere cannot cancel deterioration.

With one action per unit this retains the existing loss-only criterion. With concurrent actions, an exact MILP linearization represents mixed-sign final losses in single-objective and all multi-objective methods, including when loss is constrained, has zero weight, or is evaluated after another objective is optimized. Non-negative effects give a valid zero-loss objective.

Action and feature subsets follow `add_objective_max_benefit()`: an interaction is included only when all of its members belong to the scope. An unselected unit contributes zero loss. Use targets or other objectives if avoiding intervention altogether should not be an acceptable solution.

Value

An updated `Problem` object.

Lifecycle

Deprecated since 1.4.0. Legacy calls retain the negative-part criterion. add_objective_min_effect() is not a mathematically equivalent replacement.

Repeated calls

With `alias = NULL`, one explicit single objective can be defined per problem; a second unaliased definition raises an error. With an alias, objectives accumulate under distinct names; a repeated alias raises an error. Aliased definitions preserve an explicitly configured single objective. To compare alternatives, start from the problem before its objective was added.

See Also

add_objective_max_benefit, add_effects

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

p <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  )

p1 <- add_objective_min_loss(p)
p1$data$model_args

p2 <- add_objective_min_loss(
  p,
  actions = "restore"
)
p2$data$model_args

p3 <- add_objective_min_loss(
  p,
  features = 1
)
p3$data$model_args
```

`add_profit` *Add profit to a planning problem*

Description

Assign economic returns to feasible planning-unit–action pairs. Returns may be positive, zero, or negative and are stored separately from implementation costs and ecological effects.

Usage

```
add_profit(x, profit = NULL)
```

Arguments

`x`	A `Problem` object created with `create_problem`. It must already contain feasible actions and an action catalogue; run `add_actions` first.
`profit`	Profit specification. One of: • `NULL`: profit is set to 0 for all feasible `(pu, action)` pairs, • a numeric scalar: recycled to all feasible pairs, • a named numeric vector: names are action ids and values define action-level profit, • a `data.frame(action, profit)` defining action-level profit, • a `data.frame(pu, action, profit)` defining pair-specific profit. Profit can be defined only once per problem, including explicit zero profit. A second call raises an error. To compare profit scenarios, build separate problems from the object before profit was added.

Details

Economic returns and implementation costs

An economic return π_{ia} is the monetary contribution associated with selecting action a in planning unit i. Positive values represent gains or revenues; negative values represent penalties or economic losses. A zero value contributes no profit.

Profit is registered independently of planning-unit and action costs. `add_objective_max_profit()` maximises the sum of registered returns, whereas `add_objective_max_net_profit()` can subtract implementation costs. If your profit inputs already account for these costs, avoid subtracting them again when configuring the objective. Economic returns should not be confused with the signed ecological changes registered using `add_effects()`.

Supported input formats

The `profit` argument accepts:

- `NULL`: zero profit for all feasible pairs;
- one numeric value: the same profit for every feasible pair;
- a named numeric vector: one profit value per named action;

- `data.frame(action, profit)`: one profit value per action;
- `data.frame(pu, action, profit)`: values specific to individual planning-unit–action pairs.

An action-level value applies to all feasible planning units for that action. A pair-specific table permits spatially varying returns. Feasible pairs not specified in the input receive profit zero.

Storage and use in planning

Only non-zero values are stored in `dist_profit`. Zero-profit pairs are omitted from the table but remain feasible. The stored columns are `pu`, `action`, `profit`, `internal_pu`, and `internal_action`. Calling `add_profit()` neither selects actions nor modifies feasible actions or ecological effects; the values are used later by objectives, constraints, and solution summaries.

Profit can be defined only once per problem, including when supplying `NULL`. To compare economic scenarios, start from the same problem before profit was added.

Value

An updated `Problem` object with a stored profit table created. The stored table contains columns `pu`, `action`, `profit`, `internal_pu`, and `internal_action`, and includes only rows with non-zero profit.

See Also

`add_actions`, `add_objective_max_profit`, `add_objective_max_net_profit`, `add_effects`

Examples

```
# EXAMPLE 1: Create a planning problem with feasible actions

# The bundled 64-unit landscape has protection and restoration
# alternatives. Action implementation costs are registered separately
# from the economic returns introduced below.

sim <- load_sim_multiaction()

p <- create_problem(
  pu = sim$planning_units,
  features = sim$features,
  dist_features = sim$dist_features,
  cost = "cost"
) |>
  add_actions(
    actions = sim$actions,
    cost = sim$action_costs
  )

# EXAMPLE 2: A constant economic return

# Suppose implementing any feasible action generates 10 monetary units,
# regardless of the action or its location. A scalar is applied to all
# feasible planning-unit--action pairs.
```

```
p_constant <- add_profit(p, profit = 10)
head(p_constant$data$dist_profit[, c("pu", "action", "profit")])

# EXAMPLE 3: Action-specific returns

# Protection generates +50 monetary units, whereas restoration incurs
# an economic penalty of -5 units. A named vector assigns a value to
# every feasible planning unit where that action is available.

returns <- c(protect = 50, restore = -5)
p_action <- add_profit(p, profit = returns)
head(p_action$data$dist_profit[, c("pu", "action", "profit")])

# The same action-level values can be supplied as a data frame.
# These alternatives start from the same base problem because profit
# can be defined only once per problem.

p_action_table <- add_profit(p, data.frame(
  action = c("protect", "restore"),
  profit = c(50, -5)
))
head(p_action_table$data$dist_profit[, c("pu", "action", "profit")])

# EXAMPLE 4: Location-specific economic returns

# The same action can have different returns across planning units.
# Protection provides +100 in unit 1 and +80 in unit 2; restoration
# provides +30 in unit 3. Unlisted feasible pairs receive zero profit.

local_returns <- data.frame(
  pu = c(1, 2, 3),
  action = c("protect", "protect", "restore"),
  profit = c(100, 80, 30)
)
p_local <- add_profit(p, profit = local_returns)

# Only the three non-zero contributions are stored. This does not
# make any of the other, zero-profit action pairs infeasible.
p_local$data$dist_profit[, c("pu", "action", "profit")]

# EXAMPLE 5: Explicitly specify no economic return

# NULL assigns zero to every feasible pair. The resulting sparse
# profit table is empty, even though the actions remain available.

p_zero <- add_profit(p, profit = NULL)
nrow(p_zero$data$dist_profit)

# EXAMPLE 6: Map spatially varying potential returns

# The simulated planning units include geometry and an x-coordinate.
# Give restoration a hypothetical return that increases from west to
# east. These are potential returns, not selected management actions.
```

```
# No optimiser or solver is needed to visualise the input values.

if (requireNamespace("sf", quietly = TRUE)) {
  spatial_returns <- data.frame(
    pu = sim$planning_units$id,
    action = "restore",
    profit = 20 + 5 * sim$planning_units$x
  )

  p_spatial <- add_profit(p, profit = spatial_returns)

  # Attach the registered restoration returns to the geometry.
  values <- subset(
    p_spatial$data$dist_profit,
    action == "restore",
    select = c("pu", "profit")
  )
  mapped <- merge(
    sim$planning_units,
    values,
    by.x = "id", by.y = "pu", all.x = TRUE
  )

  # This map describes where restoration could generate economic
  # returns, not where optimisation has selected restoration.
  plot(mapped["profit"],
       main = "Potential economic return from restoration")
}
```

`add_spatial_boundary` *Add spatial boundary-length relations*

Description

Build and register a boundary-length spatial relation between planning units.

Boundary relations represent shared edge length between adjacent polygons. In contrast to queen adjacency, they only account for boundary segments of positive length and ignore point-only contacts.

Usage

```
add_spatial_boundary(
  x,
  boundary = NULL,
  geometry = NULL,
  name = "boundary",
  weight_col = NULL,
  weight_multiplier = 1,
```

```
  include_self = TRUE,
  edge_factor = 1
)
```

Arguments

x	A Problem object.
boundary	Optional data.frame describing boundary lengths. Accepted formats are: • (id1, id2, boundary), or • (pu1, pu2, weight).
geometry	Optional sf object with planning-unit polygons and an id column. If NULL, x$data$pu_sf is used.
name	Character string giving the key under which the relation is stored.
weight_col	Optional character string giving the name of the weight column in boundary. If NULL, the function tries to infer it from "boundary" or "weight".
weight_multiplier	Positive numeric scalar applied to all boundary weights.
include_self	Logical. If TRUE, include diagonal entries representing exposed boundary.
edge_factor	Numeric scalar greater than or equal to zero. Multiplier applied to exposed boundary when constructing diagonal entries.

Details

Use this function when spatial structure should be represented through shared boundary length rather than binary contiguity or coordinate-based proximity.

Two input modes are supported:

1. **Boundary-table mode.** If boundary is supplied, it is interpreted as a boundary table, for example a Marxan-style bound.dat.
2. **Geometry mode.** If boundary = NULL, boundary lengths are derived from polygon geometry using geometry or x$data$pu_sf.

Let $\omega_{ij} \geq 0$ denote the shared boundary length between planning units i and j, multiplied by weight_multiplier.

For off-diagonal entries $i \neq j$, the stored weight is:

$$\omega_{ij} = \gamma \times b_{ij},$$

where b_{ij} is the shared boundary length and γ is the user-supplied weight_multiplier.

If include_self = TRUE, diagonal entries are also created. These are not geometric self-neighbours in the graph sense; instead, they represent the effective boundary exposed to the outside of the solution.

Let p_i be the total perimeter of planning unit i, and let $\sum_{j \neq i} \omega_{ij}$ be the total incident shared boundary recorded for that planning unit. Then the exposed boundary is represented by a diagonal term derived from the difference between total perimeter and shared boundary, scaled by edge_factor.

These diagonal terms are useful in boundary-based compactness or fragmentation objectives, because they encode the portion of each planning unit's perimeter that would remain exposed if the unit were selected.

Boundary-table mode

If `boundary` is provided, accepted formats are:

- `(id1, id2, boundary)`, or
- `(pu1, pu2, weight)`.

If the table contains diagonal rows (i, i), these are interpreted as total perimeter values in boundary-table mode.

Geometry mode

If `boundary = NULL`, shared boundary lengths are derived directly from polygon geometry. Only positive-length intersections are retained. Point touches are ignored.

Storage

The final relation is stored through `add_spatial_relations`, typically as an undirected relation with optional diagonal entries.

Value

An updated `Problem` object with the stored relation in `x$data$spatial_relations[[name]]`.

See Also

`add_spatial_relations`, `add_objective_min_fragmentation_pu`, `add_objective_min_fragmentation_action`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

p <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
)

p <- add_spatial_boundary(
  x = p,
  name = "boundary",
  include_self = TRUE,
  edge_factor = 1
)

p$data$spatial_relations$boundary
```

`add_spatial_distance` *Add distance-threshold spatial relations*

Description

Build and register a spatial relation connecting planning units whose Euclidean distance is less than or equal to a user-defined threshold.

This constructor does not require polygon geometry and instead uses planning-unit coordinates.

Usage

```
add_spatial_distance(
  x,
  coords = NULL,
  max_distance,
  name = "distance",
  weight_mode = c("constant", "inverse", "inverse_sq"),
  distance_eps = 1e-09
)
```

Arguments

`x`	A `Problem` object created with `create_problem`.
`coords`	Optional coordinates specification, following the same rules as in `add_spatial_knn`.
`max_distance`	Positive numeric scalar giving the maximum distance for an edge.
`name`	Character string giving the key under which the relation is stored.
`weight_mode`	Character string indicating how distance is converted to weight. Must be one of `"constant"`, `"inverse"`, or `"inverse_sq"`.
`distance_eps`	Small positive numeric constant used to avoid division by zero in inverse-distance weighting.

Details

Use this function when neighbourhood should be defined by a fixed distance radius rather than by polygon topology or a fixed number of neighbours.

Let $s_i = (x_i, y_i)$ denote the coordinates of planning unit i. Let d_{ij} be the Euclidean distance between planning units i and j.

For a user-supplied threshold $d_{\max}$, this constructor creates an edge between i and j whenever:

$$d_{ij} \leq d_{\max}.$$

Edge weights are assigned according to `weight_mode`:

- `"constant"`:

$$\omega_{ij} = 1,$$

- `"inverse"`:

$$\omega_{ij} = \frac{1}{\max(d_{ij}, \varepsilon)},$$

- `"inverse_sq"`:

$$\omega_{ij} = \frac{1}{\max(d_{ij}, \varepsilon)^2},$$

where ε = `distance_eps` is a small constant.

The implementation computes an $O(n^2)$ distance matrix and is therefore best suited to small or moderate numbers of planning units. For large problems, `add_spatial_knn` is often more scalable.

The resulting relation is registered as undirected.

Value

An updated `Problem` object.

See Also

`add_spatial_knn`, `add_spatial_relations`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

p <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
)

p <- add_spatial_distance(
  x = p,
  max_distance = 1.01,
  name = "within_1",
  weight_mode = "constant"
)

p$data$spatial_relations$within_1
```

`add_spatial_knn` *Add k-nearest-neighbours spatial relations*

Description

Build and register a k-nearest-neighbours graph between planning units using coordinates.

This constructor does not require polygon geometry. It uses planning-unit coordinates supplied explicitly or stored in the `Problem` object.

Usage

```
add_spatial_knn(
  x,
  coords = NULL,
  k = 8,
  name = "knn",
  weight_mode = c("constant", "inverse", "inverse_sq"),
  distance_eps = 1e-09
)
```

Arguments

`x`	A `Problem` object created with `create_problem`.
`coords`	Optional coordinates specification. This may be: • a `data.frame(id, x, y)`, or • a numeric matrix with two columns `(x, y)` aligned to the order of planning units. If `NULL`, coordinates are taken from `x$data$pu_coords` or from columns `x$data$pu$x` and `x$data$pu$y`.
`k`	Integer giving the number of neighbours per planning unit. Must be at least 1 and strictly less than the number of planning units.
`name`	Character string giving the key under which the relation is stored.
`weight_mode`	Character string indicating how distance is converted to weight. Must be one of `"constant"`, `"inverse"`, or `"inverse_sq"`.
`distance_eps`	Small positive numeric constant used to avoid division by zero in inverse-distance weighting.

Details

Use this function when neighbourhood should be defined by a fixed number of nearby planning units rather than by polygon topology or a fixed distance threshold.

Let $s_i = (x_i, y_i)$ denote the coordinates of planning unit i. For each planning unit, this function identifies the k nearest distinct planning units under Euclidean distance.

If d_{ij} denotes the Euclidean distance between units i and j, then the k-nearest-neighbours relation is constructed by adding an edge from i to each of its k nearest neighbours.

Edge weights are then assigned according to `weight_mode`:

- `"constant"`:
$$\omega_{ij} = 1,$$
- `"inverse"`:
$$\omega_{ij} = \frac{1}{\max(d_{ij}, \varepsilon)},$$
- `"inverse_sq"`:
$$\omega_{ij} = \frac{1}{\max(d_{ij}, \varepsilon)^2},$$

where ε = `distance_eps` is a small constant to avoid division by zero.

The raw k-nearest-neighbours structure is directional by construction, but the stored relation is registered as undirected by default through `add_spatial_relations`, which collapses duplicate unordered pairs.

If the **RANN** package is available, it is used for efficient nearest neighbour search. Otherwise, a full distance matrix is computed.

Value

An updated `Problem` object.

See Also

`add_spatial_distance`, `add_spatial_relations`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

p <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
)

p <- add_spatial_knn(
  x = p,
  k = 2,
  name = "knn2",
  weight_mode = "constant"
)

p$data$spatial_relations$knn2
```

`add_spatial_queen` *Add queen adjacency from polygons*

Description

Build and register a queen adjacency relation from planning-unit polygons.

Two planning units are queen-adjacent if their boundaries touch, either along a shared edge or at a shared vertex.

Usage

```
add_spatial_queen(x, geometry = NULL, name = "queen", weight = 1)
```

Arguments

`x`	A `Problem` object created with `create_problem` or another object containing aligned planning-unit polygons.
`geometry`	Optional `sf` object with planning-unit polygons and an `id` column. If `NULL`, `x$data$pu_sf` is used.
`name`	Character string giving the key under which the relation is stored.
`weight`	Numeric scalar giving the edge weight assigned to each queen adjacency.

Details

Use this function when neighbourhood should include both shared edges and corner-touching polygon contacts.

This constructor derives an adjacency graph from polygon geometry using a queen criterion. If planning units i and j touch at any boundary point, then an edge (i, j) is added to the relation.

Let $G = (\mathcal{I}, E)$ denote the resulting graph. Then:

$$(i, j) \in E \quad \Longleftrightarrow \quad \partial i \cap \partial j \neq \varnothing.$$

Thus, queen adjacency includes all rook neighbours plus corner-touching neighbours.

All edges receive the same user-supplied weight.

The resulting relation is stored as an undirected spatial relation.

Value

An updated `Problem` object.

See Also

`add_spatial_rook`, `add_spatial_boundary`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

p <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
)

p <- add_spatial_queen(
  x = p,
  geometry = example_data$planning_units,
  name = "queen",
  weight = 1
)

head(p$data$spatial_relations$queen)
```

---

add_spatial_relations *Add spatial relations*

Description

Register an externally computed spatial relation inside a `Problem` object using the unified internal representation adopted by `multiscape`.

Most users will typically prefer one of the convenience constructors such as `add_spatial_boundary`, `add_spatial_rook`, `add_spatial_queen`, `add_spatial_knn`, or `add_spatial_distance`. This function is the advanced low-level entry point for adding an already computed relation.

Usage

```
add_spatial_relations(x, relations, name, directed = FALSE, allow_self = FALSE)
```

Arguments

`x`	A `Problem` object created with `create_problem`.
`relations`	A `data.frame` describing relation edges. It must contain either: • `pu1`, `pu2`, and `weight`, using external planning-unit ids, or • `internal_pu1`, `internal_pu2`, and `weight`, using internal planning-unit indices. Extra columns such as `distance` or `source` are allowed and are preserved when possible.
`name`	Character string giving the key under which the relation is stored.

`directed` Logical. If `FALSE`, treat rows as undirected edges; each unordered pair must occur exactly once. If `TRUE`, keep rows as directed ordered pairs; reciprocal arcs are distinct, but duplicate arcs are rejected.

`allow_self` Logical. If `TRUE`, allow diagonal entries (i, i). In fragmentation objectives, diagonal entries contribute as unary terms associated with the corresponding planning unit. Default is `FALSE`.

Details

Use this function when the spatial relation has already been computed externally and should be registered directly in the `Problem` object.

The input relation may be provided either in terms of external planning-unit identifiers or in terms of internal planning-unit indices.

Specifically, the input `relations` table must contain either:

- `pu1`, `pu2`, and `weight`, or
- `internal_pu1`, `internal_pu2`, and `weight`.

If external ids are supplied, they are mapped to internal indices using the planning-unit identifiers stored in the problem.

Let $G = (\mathcal{I}, E, \omega)$ denote the supplied relation, where E corresponds to the rows of `relations`. If `directed = FALSE`, each edge is treated as undirected, so pairs (i, j) and (j, i) are interpreted as the same edge. In that case, each unordered pair must be supplied exactly once. Duplicate undirected edges are rejected, including reciprocal rows.

If `directed = TRUE`, edges are preserved as ordered pairs, so (i, j) and (j, i) are distinct unless the user provides both.

#' Self-edges (i, i) are permitted only if `allow_self = TRUE`. In fragmentation objectives, diagonal entries are interpreted as unary planning-unit terms $\omega_{ii} x_i$, including for directed relations; they are not interpreted as directed self-dependencies.

The final relation is stored in `x$data$spatial_relations[[name]]`.

Calls accumulate relations with distinct names. If a relation with the same `name` already exists, an error is raised, including through spatial constructors. To change it, rebuild from the problem before it was added.

Value

An updated `Problem` object with the relation stored in `x$data$spatial_relations[[name]]`.

See Also

`add_spatial_boundary`, `add_spatial_rook`, `add_spatial_queen`, `add_spatial_knn`, `add_spatial_distance`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()
```

```
p <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
)

rel <- data.frame(
  pu1 = c(1, 1, 2),
  pu2 = c(2, 3, 3),
  weight = c(1, 1, 2)
)

p <- add_spatial_relations(
  x = p,
  relations = rel,
  name = "my_relation"
)

p$data$spatial_relations$my_relation
```

`add_spatial_rook` *Add rook adjacency from polygons*

Description

Build and register a rook adjacency relation from planning-unit polygons.

Two planning units are rook-adjacent if they share a boundary segment of positive length. Corner-only contact does not count as rook adjacency.

Usage

```
add_spatial_rook(x, geometry = NULL, name = "rook", weight = 1)
```

Arguments

`x`	A `Problem` object created with `create_problem` or another object containing aligned planning-unit polygons.
`geometry`	Optional `sf` object with planning-unit polygons and an `id` column. If `NULL`, `x$data$pu_sf` is used.
`name`	Character string giving the key under which the relation is stored.
`weight`	Numeric scalar giving the edge weight assigned to each rook adjacency.

Details

Use this function when neighbourhood should be defined by shared polygon edges rather than by point-touching or coordinate-based proximity.

This constructor derives an adjacency graph from polygon geometry using a rook criterion. If planning units i and j share a common edge of non-zero length, then an edge (i, j) is added to the relation.

Let $G = (\mathcal{I}, E)$ denote the resulting graph. Then:

$$(i, j) \in E \quad \Longleftrightarrow \quad \text{length}(\partial i \cap \partial j) > 0.$$

All edges receive the same user-supplied weight.

The resulting relation is stored as an undirected spatial relation.

Value

An updated `Problem` object.

See Also

`add_spatial_queen`, `add_spatial_boundary`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

p <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
)

p <- add_spatial_rook(
  x = p,
  geometry = example_data$planning_units,
  name = "rook",
  weight = 1
)

head(p$data$spatial_relations$rook)
```

compile_model *Compile the optimization model stored in a Problem*

Description

Materializes the optimization model represented by a Problem object without solving it. This is an advanced function mainly intended for debugging, inspection, and explicit model preparation.

In standard workflows, users normally do not need to call this function, because solve compiles the model automatically when needed.

Usage

```
compile_model(x, force = FALSE, ...)

## S3 method for class 'Problem'
compile_model(x, force = FALSE, ...)
```

Arguments

x	A Problem object.
force	Logical. If TRUE, rebuild the model even if a current compiled model already exists.
...	Reserved for future extensions.

Details

Use this function when you want to prepare the optimization model explicitly before solving, inspect compiled model structures, or verify that the problem compiles successfully.

Conceptually, a Problem object stores a declarative optimization specification: planning data, actions, effects, targets, constraints, objectives, spatial relations, and optional method or solver settings. compile_model() transforms that stored specification into an internal compiled model representation that can later be reused by the solving layer.

The exact compiled representation is implementation-specific, but it may include indexed variables, prepared constraint blocks, objective structures, and internal model snapshots or pointers.

Compilation does not solve the optimization problem. Therefore, a problem may compile successfully and still later be infeasible, numerically difficult, or otherwise fail during solver execution.

Value

A Problem object with compiled model structures stored internally.

See Also

solve

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

x <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.3) |>
  add_objective_min_cost(alias = "cost")

x <- compile_model(x)

# Force recompilation
x <- compile_model(x, force = TRUE)
```

`create_problem` *Create a planning problem input object*

Description

Create a `Problem` object from tabular or spatial inputs.

`create_problem()` is the main entry point to the `multiscape` workflow. Its role is to standardize heterogeneous planning inputs into a common internal representation that can later be used by actions, effects, targets, constraints, spatial relations, objectives, and solvers.

In all supported workflows, the result is a `Problem` object with a canonical tabular core, optionally enriched with aligned spatial metadata such as coordinates, geometry, raster references, or raw planning-unit attributes.

Usage

```
create_problem(
  pu,
  features,
  dist_features,
  cost = NULL,
  pu_id_col = "id",
  cost_aggregation = c("mean", "sum"),
```

```
  ...
)

## S4 method for signature 'data.frame,data.frame,data.frame'
create_problem(
  pu,
  features,
  dist_features,
  cost = NULL,
  pu_id_col = "id",
  cost_aggregation = c("mean", "sum"),
  ...
)

## S4 method for signature 'ANY,ANY,missing'
create_problem(
  pu,
  features,
  dist_features,
  cost = NULL,
  pu_id_col = "id",
  cost_aggregation = c("mean", "sum"),
  ...
)

## S4 method for signature 'ANY,ANY,NULL'
create_problem(
  pu,
  features,
  dist_features,
  cost = NULL,
  pu_id_col = "id",
  cost_aggregation = c("mean", "sum"),
  ...
)

## S4 method for signature 'ANY,data.frame,data.frame'
create_problem(
  pu,
  features,
  dist_features,
  cost = NULL,
  pu_id_col = "id",
  cost_aggregation = c("mean", "sum"),
  ...
)
```

Arguments

pu	Planning-units input. Depending on the selected method, this may be: • a data.frame with planning-unit information, • an sf object, • a terra::SpatVector, • a vector file path readable by **terra**, • a terra::SpatRaster, • or a raster file path.
features	Feature input. Depending on the selected workflow, this may be: • a data.frame with at least an id column, • a terra::SpatRaster with one layer per feature, • or a raster file path.
dist_features	Feature-distribution input. In tabular and hybrid workflows, this must be a data.frame with columns pu, feature, and amount. In spatial modes it may be omitted or set to NULL, in which case it is derived automatically.
cost	In spatial modes, planning-unit cost information. Depending on the input mode, this may be: • a column name in the planning-unit attribute table, • a raster object, • or a raster file path. In raster-cell mode, cost is required and valid planning units are defined only where cost is finite and strictly positive.
pu_id_col	Character string giving the name of the planning-unit id column in vector or sf inputs. Ignored in purely tabular mode and in raster-cell mode.
cost_aggregation	Character string used in vector-PU mode when cost is a raster and must be aggregated over polygons. Must be one of "mean" or "sum".
...	Additional arguments forwarded to internal builders.

Details

Feature amounts describe a user-defined reference scenario, which can be current conditions, a future business-as-usual scenario, or existing management. The term baseline in older interfaces refers to these reference amounts. Action outcomes and reference amounts must share units and, for future scenarios, the same time horizon. The package optimizes supplied effects; it does not estimate causal effects or simulate reference scenarios.

A Problem object created by create_problem() is the basic input structure used throughout downstream multiscape workflows.

The returned object always contains a canonical tabular planning core and may additionally store aligned spatial metadata depending on the input workflow. This is the internal representation on which later functions such as add_actions(), add_effects(), add_constraint_targets(), add_spatial_relations(), and solve() operate.

Conceptual role of create_problem()

Regardless of the input workflow, `create_problem()` aims to produce a consistent internal representation containing at least:

- a planning-unit table,
- a feature table,
- a feature-distribution table.

Internally, the problem is always reduced to a tabular core of the form:

- planning units $i \in \mathcal{I}$,
- features $f \in \mathcal{F}$,
- non-negative baseline amounts $a_{if} \geq 0$,
- and, when available, spatial metadata associated with planning units.

The feature-distribution table provides the canonical sparse representation of these baseline amounts. Thus, after `create_problem()` has run, the landscape is internally represented by baseline feature amounts a_{if}, where a_{if} denotes the amount of feature f in planning unit i. These baseline amounts are later combined with actions, effects, targets, objectives, and constraints to build the optimization model.

Which input mode should I use?

`create_problem()` supports four main workflows. The best choice depends on the form of your data and on whether you want to preserve geometry.

- **Tabular mode.** Use this when `pu`, `features`, and `dist_features` are already available as `data.frame` objects and no spatial derivation is needed. This is the simplest workflow: all problem components are already tabular, so `create_problem()` only standardizes them into the canonical internal representation.
- **Vector-PU spatial mode.** Use this when planning units are polygons and feature information is stored in one or more raster layers. In this mode, `dist_features` is derived by aggregating raster values over planning-unit polygons.
- **Raster-cell fast mode.** Use this when both `pu` and `features` are rasters and each valid raster cell should become one planning unit. This mode avoids raster-to-polygon conversion, treats `NA` feature values as zero before building `dist_features`, keeps only strictly positive amounts in the stored distribution table, and is generally the preferred option for large regular grids.
- **Hybrid sf + tabular mode.** Use this when you already have a curated tabular `dist_features` table but still want to preserve planning-unit geometry and attributes for later plotting, spatial relations, or feasibility specifications. Here, the feature distribution is already tabular, but geometry and raw attributes are preserved from the `sf` planning-unit object for later spatial operations.

How cost is interpreted across modes

`cost` handling depends on the selected workflow.

- **Tabular mode.** In purely tabular mode, `cost` is not used by this generic method. Planning-unit costs are expected to already be present in the `pu` table supplied to the internal tabular builder.
- **Vector-PU spatial mode.** In vector-PU mode, `cost` is required and may be either:

  - the name of a numeric attribute column in the planning-unit layer,
  - or a cost raster to be aggregated over polygons using the `cost_aggregation` argument.

- **Raster-cell fast mode.** In raster-cell mode, `cost` must be a single-layer raster aligned with `pu` and `features`. A raster cell becomes a planning unit only if the mask cell is not missing and the corresponding cost value is finite and strictly positive. In other words, if m_i denotes the mask value of cell i and c_i its cost value, then cell i is retained only when the mask is observed and $c_i > 0$.
- **Hybrid sf + tabular mode.** In hybrid mode, `cost` must be either:
  - the name of a numeric attribute column in the `sf` layer,
  - or omitted if the `sf` attributes already contain a column literally named `cost`.

Feature identifiers

Features are internally standardized to an `id`-based representation. In spatial modes where raster layers are used, one feature is created per raster layer, with:

- `id` = `1, 2, ..., nlyr(features)`,
- `name` equal to the raster layer name, if available,
- otherwise a generated name of the form `"feature.1"`, `"feature.2"`, and so on.

Planning-unit identifiers

In vector and hybrid modes, planning-unit ids are taken from the column named by `pu_id_col`. If that column is missing and `pu_id_col = "id"`, sequential ids are created with a warning.

In raster-cell mode, planning-unit ids are always created sequentially from the valid raster cells retained after masking and cost filtering.

Ordering and alignment

In spatial modes, planning units, derived coordinates, raw attributes, geometry, and extracted feature amounts are aligned to the same planning-unit id order before the final `Problem` object is created.

This alignment is critical because later functions assume that all stored planning-unit components refer to the same ordered set of planning units.

After `create_problem()`, typical next steps include adding actions, spatial relations, targets or other constraints, objectives, and then solving the resulting problem.

Value

A `Problem` object for downstream `multiscape` workflows. The returned object always contains a canonical tabular core with planning units, features, and feature-distribution data, and may additionally contain aligned spatial metadata depending on the input mode.

See Also

`add_actions`, `add_constraint_locked_pu`, `add_spatial_boundary`, `add_spatial_knn`, `solve`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

# Spatial mode: keep planning-unit geometries.
p1 <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
)
print(p1)

# Tabular mode: use the same data without geometry.
p2 <- create_problem(
  pu = sf::st_drop_geometry(example_data$planning_units),
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
)
print(p2)
```

---

frontier_distances *Compute distances to observed ideal or nadir points*

Description

Compute normalized distances from each stored solution in a solutionset-class object to the observed ideal and/or nadir point in objective space.

Usage

```
frontier_distances(
  x,
  objectives = NULL,
  reference = "ideal",
  metric = c("euclidean", "manhattan", "chebyshev")
)
```

Arguments

x	A solutionset-class object returned by solve.
objectives	Optional character vector of objective names to use. If NULL, all available objective-value columns are used.
reference	Character vector indicating the reference point or points to use. Allowed values are "ideal" and "nadir". If both are supplied, distances and ranks for both reference points are returned. Default is "ideal".

`metric` Character. Distance metric to use. One of `"euclidean"`, `"manhattan"`, or `"chebyshev"`.

Details

This function supports the interpretation and ranking of trade-offs among solutions stored in a `SolutionSet`.

The calculations are based on the solutions contained in the supplied object. Therefore, the observed ideal point, observed nadir point, objective ranges, normalized values, and distances may change when the input `SolutionSet` is filtered.

To calculate distances using only non-dominated solutions, first use:

```
x_nd <- solution_filter(x, nondominated = TRUE)
frontier_distances(x_nd)
```

Objective values are internally transformed to a common minimization space using the objective senses registered in the original problem. `sense = "min"` are kept unchanged, whereas objectives with `sense = "max"` are multiplied by -1. In this transformed space, lower values are always better.

The observed ideal point is defined by the best observed value for each objective:

$$z_j^{ideal} = \min_i z_{ij},$$

where z_{ij} is the transformed value of solution i for objective j.

The observed nadir point is defined by the worst observed value for each objective:

$$z_j^{nadir} = \max_i z_{ij}.$$

These reference points are empirical bounds derived from the supplied solutions. They are not necessarily the true ideal and nadir points of the complete feasible objective space.

Objective values are normalized to the interval $[0, 1]$ using:

$$\tilde{z}_{ij} = \frac{z_{ij} - z_j^{ideal}}{z_j^{nadir} - z_j^{ideal}}.$$

After normalization:

- `0` represents the best observed value for an objective;
- `1` represents the worst observed value for an objective.

This interpretation is independent of whether the original objective was minimized or maximized.

If an objective has zero observed range, all normalized values for that objective are set to zero. The objective therefore does not contribute to the calculated distances.

For distances to the ideal point, smaller values are preferred and `rank_to_ideal = 1` identifies the closest solution.

For distances to the nadir point, larger values are preferred and `rank_from_nadir = 1` identifies the solution farthest from the observed nadir point.

Value

A `data.frame` with one row per stored solution having complete values for the selected objectives.

The table contains:

- `solution_id`: integer solution id;
- the original objective values;
- normalized objective values prefixed with `norm_`;
- distance and rank columns for the requested reference points.

The returned table also contains the following attributes:

- `"ideal"`: observed ideal point in the original objective scales;
- `"nadir"`: observed nadir point in the original objective scales;
- `"ranges"`: observed absolute ranges in the original objective scales;
- `"objectives"`: objective names used;
- `"sense"`: optimization sense of each objective;
- `"metric"`: distance metric used;
- `"reference"`: requested reference point or points;
- `"normalized"`: whether normalized values were used;
- `"space"`: objective space used for distance calculations.

See Also

`frontier_extremes`, `get_objectives`, `solution_filter`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
  add_objective_max_effect(alias = "benefit") |>
  set_method_weighted_sum(
```

```
  aliases = c("cost", "benefit"),
  runs = set_runs_grid(n = 3),
  normalize_weights = TRUE
)

if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(
    problem,
    verbose = FALSE
  )

  solutions <- solve(problem)

  # Normalized Euclidean distance to the observed ideal point
  frontier_distances(solutions)

  # Distances to both observed ideal and nadir points
  distances <- frontier_distances(
    solutions,
    reference = c("ideal", "nadir")
  )

  # Use only selected objectives
  frontier_distances(
    solutions,
    objectives = c("cost", "benefit")
  )

  # Use Manhattan distance
  frontier_distances(
    solutions,
    metric = "manhattan"
  )

  # Use Chebyshev distance
  frontier_distances(
    solutions,
    metric = "chebyshev"
  )

  # Inspect observed reference points and objective ranges
  attr(distances, "ideal")
  attr(distances, "nadir")
  attr(distances, "ranges")

  # Calculate distances only over non-dominated solutions
  if (requireNamespace("moocore", quietly = TRUE)) {
    nondominated_solutions <- solution_filter(
      solutions,
      feasible_only = TRUE,
      nondominated = TRUE
    )
```

```
    frontier_distances(
      nondominated_solutions,
      reference = c("ideal", "nadir")
    )
  }
}
```

`frontier_extremes` *Find objective-wise extreme solutions*

Description

Identify the observed minimum and maximum values for each objective in a `solutionset-class` object.

This function returns the solutions that define the observed range of each selected objective. It also labels each extreme as `"best"` or `"worst"` according to the registered optimization sense of the objective.

Usage

```
frontier_extremes(x, objectives = NULL, ties = c("all", "first"))
```

Arguments

`x`	A `solutionset-class` object returned by `solve`.
`objectives`	Optional character vector of objective names to inspect. If `NULL`, all available objective-value columns are used.
`ties`	Character. How to handle ties. If `"all"`, all tied solutions are returned. If `"first"`, only the first tied solution is returned.

Details

Objective values are obtained from the stored run table. Objective senses are obtained from the objective specifications stored in the original problem.

For objectives with `sense = "min"`, the observed minimum is labelled as `"best"` and the observed maximum is labelled as `"worst"`. For objectives with `sense = "max"`, the observed maximum is labelled as `"best"` and the observed minimum is labelled as `"worst"`.

Runs without a stored `solution_id` or with missing objective values for the selected objectives are ignored automatically. Therefore, infeasible runs are not considered in the computation.

If several solutions have the same extreme value for an objective, the behaviour is controlled by `ties`.

Value

A `data.frame` with one or more rows per objective. The returned columns are:

- `solution_id`: integer solution id;
- `objective`: objective name;
- `sense`: optimization sense, either `"min"` or `"max"`;
- `bound`: observed bound, either `"min"` or `"max"`;
- `role`: interpretation of the bound, either `"best"` or `"worst"`;
- `value`: objective value at the observed bound.

See Also

`get_objectives`, `solution_filter`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
  add_objective_max_effect(alias = "benefit") |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 3),
    normalize_weights = TRUE
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(
    problem,
    verbose = FALSE
  )

  solutions <- solve(problem)

  # Observed minimum and maximum for every objective
```

```
  frontier_extremes(solutions)

  # Inspect only selected objectives
  frontier_extremes(
    solutions,
    objectives = c("cost", "benefit")
  )

  # Keep only the first solution when several solutions share an extreme
  frontier_extremes(
    solutions,
    ties = "first"
  )
}
```

`frontier_knee` *Identify knee solutions on an observed Pareto frontier*

Description

Identify empirical knee, or compromise, solutions from the objective values stored in a `solutionset-class` object.

Usage

```
frontier_knee(
  x,
  objectives = NULL,
  method = c("distance", "ideal", "angle"),
  metric = c("euclidean", "manhattan", "chebyshev"),
  nondominated = TRUE,
  ties = c("first", "all"),
  return_all = FALSE
)
```

Arguments

`x`	A `solutionset-class` object returned by `solve`.
`objectives`	Optional character vector of objective names to use. If `NULL`, all available objective-value columns are used.
`method`	Character. Method used to score knee solutions. One of `"distance"`, `"ideal"`, or `"angle"`.
`metric`	Character. Distance metric used when `method = "ideal"`. One of `"euclidean"`, `"manhattan"`, or `"chebyshev"`.
`nondominated`	Logical. If `TRUE`, the knee is computed after filtering to feasible non-dominated solutions. Defaults to `TRUE`.

ties	Character. How to handle ties when return_all = FALSE. If "all", all equally ranked knee solutions are returned. If "first", only the first one is returned.
return_all	Logical. If TRUE, return all scored solutions. If FALSE, return only the highest-ranked knee solution or solutions.

Details

A knee solution is a solution located in a region of the observed frontier where small improvements in one objective tend to require relatively large losses in another objective. Because this concept can be defined in several ways, frontier_knee() provides multiple scoring methods.

Objective values are first transformed to a common minimization space using the optimization sense registered in the original problem. Objectives with sense = "min" are kept unchanged, whereas objectives with sense = "max" are multiplied by -1. Values are then normalized to the observed $[0, 1]$ range, where 0 represents the best observed value and 1 represents the worst observed value for each objective.

The available methods are:

- "distance": identifies the solution with the largest perpendicular distance to the line connecting the two observed objective-wise extreme solutions. This method requires exactly two objectives and is the default geometric knee definition.
- "ideal": identifies the solution closest to the observed ideal point in normalized objective space. This method can be used with two or more objectives and is best interpreted as a compromise solution.
- "angle": identifies the solution with the largest change in direction along the observed bi-objective frontier. This method requires exactly two objectives and at least three complete solutions.

By default, frontier_knee() first filters the supplied SolutionSet to feasible non-dominated solutions using solution_filter. Set nondominated = FALSE to compute the knee over all stored solutions with complete objective values.

The returned knee is empirical and depends on the supplied SolutionSet. It should not be interpreted as the unique knee of the full feasible objective space unless the supplied solutions adequately represent the frontier.

Value

A data.frame. If return_all = FALSE, the table contains the selected knee solution or solutions. If return_all = TRUE, the table contains all candidate solutions ranked by knee score.

The returned table includes:

- solution_id: integer solution id;
- the original objective values;
- normalized objective values prefixed with norm_;
- method-specific diagnostic columns;
- knee_score: score used to rank knee solutions, where larger values indicate stronger knee candidates;

- `knee_rank`: rank of each solution according to `knee_score`;
- `method`: knee scoring method used.

The returned table also contains the following attributes:

- `"ideal"`: observed ideal point in the original objective scales;
- `"nadir"`: observed nadir point in the original objective scales;
- `"ranges"`: observed absolute ranges in the original objective scales;
- `"objectives"`: objective names used;
- `"sense"`: optimization sense of each objective;
- `"method"`: knee method used;
- `"nondominated"`: whether non-dominated filtering was applied;
- `"space"`: objective space used for scoring.

See Also

`frontier_distances`, `frontier_extremes`, `get_objectives`, `solution_filter`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
  add_objective_max_effect(alias = "benefit") |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 3),
    normalize_weights = TRUE
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(
    problem,
    verbose = FALSE
  )
```

```
  solutions <- solve(problem)

  # Default geometric knee
  frontier_knee(solutions)

  # Return all solutions ranked by knee score
  frontier_knee(
    solutions,
    return_all = TRUE
  )

  # Closest solution to the observed ideal point
  frontier_knee(
    solutions,
    method = "ideal"
  )

  # Largest change in direction along the bi-objective frontier
  frontier_knee(
    solutions,
    method = "angle"
  )
}
```

`frontier_neighbors` *Identify neighboring solutions in objective space*

Description

Connect nearby stored solutions in normalized objective space. The resulting pairs can be supplied directly to objective–decision linkage functions.

Usage

```
frontier_neighbors(
  x,
  objectives = NULL,
  method = c("auto", "sequence", "mst", "knn"),
  metric = c("euclidean", "manhattan", "chebyshev"),
  k = 1L
)
```

Arguments

`x`	A `solutionset-class` object returned by `solve`.
`objectives`	Optional character vector of objective aliases. If `NULL`, all registered objectives are used.

method	Neighborhood method: "auto", "sequence", "mst", or "knn".
metric	Objective-space distance metric: "euclidean", "manhattan", or "chebyshev".
k	Number of nearest neighbors used only by method = "knn".
	The first selected objective orients all pairs and, for method = "sequence", also orders the solutions. Supply the desired ordering objective first in objectives.

Details

Objective values are oriented to minimization and normalized using the solutions retained in the supplied SolutionSet. The function does not remove dominated or repeated solutions; use solution_filter or solution_unique beforehand when required.

method = "auto" uses a sequence for one or two objectives and a minimum spanning tree for higher-dimensional objective spaces.

Value

A data.frame with from_solution, to_solution, objective_distance, and objective-specific values and changes. Analysis settings are retained as attributes.

See Also

frontier_distances, linkage_distances, linkage_turnover

Examples

```
# Load a complete simulated multi-action problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(
    alias = "cost",
    include_pu_cost = FALSE
  ) |>
  add_objective_max_effect(
    alias = "benefit"
  ) |>
  set_method_weighted_sum(
```

```
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 5),
    normalize_weights = TRUE
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(problem, verbose = FALSE)
  solutions <- solve(problem)

  neighbors <- frontier_neighbors(
    solutions,
    objectives = c("benefit", "cost")
  )

  neighbors

  frontier_neighbors(
    solutions,
    objectives = c("cost", "benefit"),
    method = "knn",
    k = 2
  )
}
```

`get_action_sets` *Inspect registered action sets*

Description

Return the definitions registered by `add_action_sets()` in long format. This function inspects a planning problem, not selected actions in a solution.

Usage

```
get_action_sets(x)
```

Arguments

`x`	A `Problem` object.

Value

A `data.frame` with character columns `set` and `action`, sorted by set and action id. If no sets are registered, a zero-row table with the same columns is returned. Internal identifiers are not exposed.

See Also

`add_action_sets()`, `get_actions()`

Examples

```
problem <- create_problem(
  pu = data.frame(id = 1:2, cost = 1),
  features = data.frame(id = 1L, name = "woodland"),
  dist_features = data.frame(pu = 1:2, feature = 1L, amount = 10)
) |>
  add_actions(data.frame(id = c("restore", "control")))

get_action_sets(problem)
problem <- add_action_sets(
  problem, list(restore_control = c("restore", "control"))
)
get_action_sets(problem)
```

---

get_actions *Get action results from a solution set*

Description

Extract the action-allocation summary table from a `solutionset-class` object returned by `solve`.

The returned table summarizes solution values at the planning unit–action level and typically includes a `selected` indicator showing whether each feasible `(pu, action)` pair is selected in a solution.

Usage

```
get_actions(x, solution = NULL, ...)
```

Arguments

`x`	A `solutionset-class` object returned by `solve`.
`solution`	Optional positive integer giving the solution id to extract. If `NULL`, all runs are returned when available.
`...`	Deprecated arguments kept for backwards compatibility. Currently supports `run` and `solution_id`, which are redirected to `solution`.

Details

This function reads the action summary stored in `x$summary$actions`. It does not reconstruct the table from the raw decision vector; it simply returns the stored summary after optional run filtering.

Let x_{ia} denote the decision variable associated with selecting action a in planning unit i. In standard **multiscape** workflows, the `selected` column is the user-facing representation of that decision, typically coded as `0` or `1`.

If `solution` is provided, only rows belonging to that solution are returned. This requires the summary table to contain a `solution_id` column.

To return only selected action allocations, filter the returned table using `selected == 1`.

Value

A `data.frame` containing the stored action-allocation summary. Typical columns include planning-unit ids, action ids, optional labels, and a `selected` indicator.

See Also

get_planning_units, get_features, get_targets

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
  add_objective_max_effect(alias = "benefit") |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 3),
    normalize_weights = TRUE
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(
    problem,
    verbose = FALSE
  )

  solutions <- solve(problem)

  # All feasible planning-unit/action assignments
  get_actions(solutions)

  # Only selected action assignments
  selected_actions <- get_actions(solutions)
  selected_actions <- selected_actions[
    selected_actions$selected == 1L,
    ,
    drop = FALSE
```

```
  ]
  selected_actions

  # Action allocations for one solution
  solution_ids <- get_runs(solutions)$solution_id

  get_actions(
    solutions,
    solution = solution_ids[1]
  )
}
```

`get_features` *Get feature summary from a solution set*

Description

Extract the per-feature summary table from a `solutionset-class` object returned by `solve`.

The returned table summarizes, for each feature, how much of the feature was available in the full baseline landscape and how much is represented by the selected planning units or selected actions in each run.

Usage

```
get_features(x, solution = NULL, ...)
```

Arguments

`x`	A `solutionset-class` object returned by `solve`.
`solution`	Optional positive integer giving the solution id to extract. If `NULL`, all runs are returned when available.
`...`	Deprecated arguments kept for backwards compatibility. Currently supports `run` and `solution_id`, which are redirected to `solution`.

Details

This function reads the feature summary stored in `x$summary$features`. It errors if that table is missing.

Feature summaries distinguish between baseline availability in the full landscape and the contribution of selected planning units or selected actions.

Let B_f denote the total baseline amount of feature f available in the full landscape. Let S_f denote the baseline amount of feature f in selected rows. Let A_f denote the after-action amount of feature f contributed by those selected rows. Let G_f and L_f denote the positive and negative net-change components induced by selected actions. Then:

$$\text{selected_net}_f = G_f - L_f,$$

and:

$$A_f = S_f + \text{selected_net}_f.$$

The main returned columns are:

- `baseline_total`: total baseline amount in the full landscape;
- `selected_baseline`: baseline amount in selected rows;
- `selected_amount_after`: after-action amount contributed by selected rows;
- `selected_benefit`: positive net-change component from selected actions;
- `selected_loss`: negative net-change component from selected actions;
- `selected_net`: net change from selected actions;
- `selected_fraction_of_baseline`: ratio between `selected_amount_after` and `baseline_total`.

Importantly, this summary does not assume that planning units without a selected action contribute to the achieved feature amount. Therefore, the achieved amount for a feature is represented by `selected_amount_after`, not by a full-landscape total obtained by adding net changes to the baseline.

For backwards compatibility with older result objects, if the newer selected-action columns are missing, this function attempts to construct them from older columns such as `total_available`, `benefit`, `loss`, `net`, and `amount_after`. However, the returned table is organized using the newer selected-action terminology.

If `solution` is provided, only rows belonging to that solution are returned. If the result contains a `solution_id` column but only a single solution is present and `solution` was not requested explicitly, the `solution_id` column is removed for convenience.

Positive gains and losses are split after combining actions within each unit and feature. With matching action/feature scopes, the benefit objective corresponds to the sum of `selected_net`, and the loss objective to the sum of `selected_loss`. `selected_benefit` remains a descriptive positive-gain component and is not the signed benefit objective.

This function summarizes feature outcomes in the result. It is different from `get_targets`, which focuses on target achievement.

Value

A `data.frame` with one row per feature, or one row per feature–run combination when multiple runs are present. The returned table includes, when available or derivable, the columns `baseline_total`, `selected_baseline`, `selected_amount_after`, `selected_benefit`, `selected_loss`, `selected_net`, and `selected_fraction_of_baseline`.

See Also

`get_planning_units`, `get_actions`, `get_targets`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
  add_objective_max_effect(alias = "benefit") |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 3),
    normalize_weights = TRUE
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(
    problem,
    verbose = FALSE
  )

  solutions <- solve(problem)

  # Feature outcomes for all stored runs
  get_features(solutions)

  # Feature outcomes for one run
  solution_ids <- get_runs(solutions)$solution_id

  get_features(
    solutions,
    solution = solution_ids[1]
  )
}
```

`get_objectives` *Get objective values from a solution set*

Description

Extract objective values from the runs stored in a solutionset-class object.

Usage

```
get_objectives(x, format = c("wide", "long"))
```

Arguments

x	A solutionset-class object returned by solve.
format	Character. Output representation, either "long" or "wide". Defaults to "wide".

Details

Objective values are read from run-table columns named value_<objective>, where <objective> is the registered objective alias.

Runs without a stored solution may contain missing objective values. Filter the SolutionSet beforehand with solution_filter when only solved runs should be included.

Public objective tables are keyed by integer solution_id. Use get_runs when the relationship between attempted run_ids and stored solutions is required. In long format, every solution-objective combination occupies one row. In wide format, every stored solution occupies one row and every objective occupies one column.

Value

If format = "long", a data.frame with columns solution_id, objective, and value.

If format = "wide", a data.frame with integer solution_id and one column per objective.

See Also

get_runs, frontier_extremes, frontier_distances

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
```

```
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
  add_objective_max_effect(alias = "benefit") |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 3),
    normalize_weights = TRUE
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(
    problem,
    verbose = FALSE
  )

  solutions <- solve(problem)

  # Long format
  get_objectives(
    solutions,
    format = "long"
  )

  # Wide format
  get_objectives(
    solutions,
    format = "wide"
  )

  # Objective values from usable runs only
  usable_solutions <- solution_filter(
    solutions,
    feasible_only = TRUE
  )
  get_objectives(usable_solutions)
}
```

`get_planning_units` *Get planning-unit results from a solution set*

Description

Extract the planning-unit summary table from a `solutionset-class` object returned by `solve`.

The returned table summarizes solution values at the planning-unit level and typically includes a `selected` indicator showing whether each planning unit is selected in a solution.

Usage

```
get_planning_units(x, solution = NULL, ...)
```

Arguments

x	A solutionset-class object returned by solve.
solution	Optional positive integer giving the solution id to extract. If NULL, all solutions are returned when available.
...	Deprecated arguments kept for backwards compatibility. Currently supports run and solution_id, which are redirected to solution.

Details

This function reads the planning-unit summary stored in x$summary$pu. It does not reconstruct the table from the raw decision vector; it simply returns the stored summary after optional run filtering.

Let w_i denote the planning-unit selection variable for planning unit i. In standard **multiscape** workflows, the selected column is the user-facing representation of that planning-unit decision, typically coded as 0 or 1.

If solution is provided, only rows belonging to that solution are returned. This requires the summary table to contain a solution_id column.

To return only selected planning units, filter the returned table using selected == 1.

Value

A data.frame containing the stored planning-unit summary. Typical columns include planning-unit identifiers, optional labels, and a selected indicator.

See Also

get_actions, get_features, get_targets

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
  add_objective_max_effect(alias = "benefit") |>
  set_method_weighted_sum(
```

```
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 3),
    normalize_weights = TRUE
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(
    problem,
    verbose = FALSE
  )

  solutions <- solve(problem)

  # Planning-unit results for all stored runs
  get_planning_units(solutions)

  # Return only selected planning units
  selected_pu <- get_planning_units(solutions)
  selected_pu <- selected_pu[selected_pu$selected == 1L, , drop = FALSE]
  selected_pu

  # Extract one run using its solution_id
  solution_ids <- get_runs(solutions)$solution_id

  get_planning_units(
    solutions,
    solution = solution_ids[1]
  )
}
```

`get_pu` *Get planning-unit results from a solution set*

Description

[Deprecated]

`get_pu()` has been replaced by `get_planning_units`.

Usage

```
get_pu(x, solution = NULL, ...)
```

Arguments

`x`	A `solutionset-class` object returned by `solve`.
`solution`	Optional positive integer giving the solution id to extract. If `NULL`, all solutions are returned when available.
`...`	Deprecated arguments kept for backwards compatibility. Currently supports `run` and `solution_id`, which are redirected to `solution`.

Value

A data.frame containing the stored planning-unit summary.

---

get_runs *Get run-level metadata from a solution set*

Description

Extract the run table from a solutionset-class object.

Usage

```
get_runs(x)
```

Arguments

x	A solutionset-class object returned by solve.

Details

A run represents an attempted optimization solve. Each run has a unique run_id. Only runs that produce a stored solution receive a solution_id.

The solution_id is numeric and matches the corresponding run_id. Therefore, if a run fails or is infeasible, its solution_id is NA; if a later run succeeds, its solution_id keeps the same value as its run_id.

This function is the user-facing place where the relationship between attempted runs and stored solutions is reported.

Objective values are not returned by get_runs(). To extract objective values, use get_objectives.

Value

A data.frame with one row per attempted optimization run. The table contains run metadata and the numeric mapping between run_id and solution_id, but not objective-value columns.

See Also

get_objectives, solution_filter, set_runs_grid, set_runs_manual

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
```

```
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
  add_objective_max_effect(alias = "benefit") |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 3),
    normalize_weights = TRUE
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(
    problem,
    verbose = FALSE
  )

  solutions <- solve(problem)

  get_runs(solutions)
}
```

`get_solution_states` *Get planning-unit states from stored solutions*

Description

Convert stored planning-unit/action decisions into one canonical action state per planning unit and solution.

Usage

```
get_solution_states(x)
```

Arguments

x	A `solutionset-class` object returned by `solve`.

Details

Selected action names are sorted and joined with `"+"`. Planning units without a selected action receive the state `"unmanaged"`. The `selected` column retains the planning-unit selection indicator when it is available in the stored planning-unit summary.

Value

A `data.frame` with columns `solution_id`, `pu`, `selected`, `state`, `managed`, and `n_actions`. `solution_id` is returned as an integer.

See Also

`get_actions`, `get_planning_units`, `selection_consistency`, `linkage_transition`

Examples

```
# Load a complete simulated multi-action problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(
    alias = "cost",
    include_pu_cost = FALSE
  ) |>
  add_objective_max_effect(
    alias = "benefit"
  ) |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 5),
    normalize_weights = TRUE
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(problem, verbose = FALSE)
  solutions <- solve(problem)

  states <- get_solution_states(solutions)
  head(states)

  # Inspect one solution from the returned table.
  solution_ids <- unique(states$solution_id)
  states[states$solution_id == solution_ids[1L], ]

  # Or filter the complete SolutionSet first.
```

```
  one_solution <- solution_filter(
    solutions,
    solution_id = solution_ids[1L]
  )
  get_solution_states(one_solution)
}
```

`get_targets` *Get target achievement summary from a solution set*

Description

Extract a user-facing target-achievement table from a `solutionset-class` object returned by `solve`.

The returned table summarizes, for each stored target, the target level, the achieved value, the gap between achieved and required values, and whether the target was met in each run.

Usage

```
get_targets(x, solution = NULL, ...)
```

Arguments

`x`	A `solutionset-class` object returned by `solve`.
`solution`	Optional positive integer giving the solution id to extract. If `NULL`, all runs are returned when available.
`...`	Deprecated arguments kept for backwards compatibility. Currently supports `run` and `solution_id`, which are redirected to `solution`.

Details

Targets are optional in **multiscape**. If the result object does not contain a targets summary table at `x$summary$targets`, this function returns `NULL` without error.

This function reads the stored targets summary and returns a simplified user-facing table. If the summary contains `achieved` and `target_value`, target satisfaction is evaluated as follows.

For lower-bound targets:

$$\text{met} = (\text{achieved} \geq \text{target}),$$

and for upper-bound targets:

$$\text{met} = (\text{achieved} \leq \text{target}).$$

The interpretation of the target direction is taken from the `sense` column when available:

- `"ge"`, `">="`, or `"min"` are treated as lower-bound targets;
- `"le"`, `"<="`, or `"max"` are treated as upper-bound targets;
- if `sense` is missing, the target is treated as a lower bound by default.

The returned table is simplified and renames some internal fields for readability:

- `target_raw` is returned as `target_level`;
- `basis_total` is returned as `total_available`;
- `target_value` is returned as `target`.

If `solution` is provided, only rows belonging to that solution are returned. If the result contains a `run_id` column but only a single solution is present and `solution` was not requested explicitly, the `solution_id` column is removed for convenience.

The gap column is expected to be part of the stored summary. When present, it typically represents:

$$\text{gap} = \text{achieved} - \text{target}.$$

Value

A simplified `data.frame` target summary, or `NULL` if the result does not contain targets. Typical columns include `feature`, `feature_name`, `target_level`, `total_available`, `target`, `achieved`, `gap`, and `met`.

See Also

`get_planning_units`, `get_actions`, `get_features`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
  add_objective_max_effect(alias = "benefit") |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 3),
    normalize_weights = TRUE
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
```

```
  problem <- set_solver_cbc(
    problem,
    verbose = FALSE
  )

  solutions <- solve(problem)

  # Target requirements and achieved amounts
  get_targets(solutions)

  # Target achievement for one run
  solution_ids <- get_runs(solutions)$solution_id

  get_targets(
    solutions,
    solution = solution_ids[1]
  )
}
```

`linkage_contrasts` *Select informative solution contrasts*

Description

Rank pairwise linkage results to identify solution pairs that are especially informative in objective space, decision space, or their relationship.

Usage

```
linkage_contrasts(
  x,
  type = c("objective_similar", "decision_similar", "high_turnover",
    "high_reconfiguration", "low_reconfiguration", "objective_tie"),
  n = 3L
)
```

Arguments

`x`	A `data.frame` returned by `linkage_distances` or `linkage_turnover`.
`type`	Contrast to rank: `"objective_similar"`, `"decision_similar"`, `"high_turnover"`, `"high_reconfiguration"`, `"low_reconfiguration"`, or `"objective_tie"`.
`n`	Number of contrasts to return.

Details

"objective_similar" ranks pairs by increasing objective distance and uses larger decision distance as a secondary criterion. Conversely, "decision_similar" ranks pairs by increasing decision distance and then decreasing objective distance. "high_turnover" ranks pairs by decreasing decision distance.

The "high_reconfiguration" and "low_reconfiguration" types require a numeric reconfiguration_rate column, as returned by linkage_turnover. Rows with an undefined reconfiguration_rate are excluded from these rankings.

"objective_tie" returns objective-equivalent pairs with non-zero decision distance, ranked from largest to smallest spatial difference. This type requires the logical objective_tie column returned by linkage_turnover. The numerical tolerance used to identify objective ties is therefore controlled only by linkage_turnover().

from_solution and to_solution must contain positive integer solution ids. Numeric values such as 1 are accepted when they are integer-valued and are returned as integers.

Value

The first n ranked rows. The columns in x are preserved, with contrast_rank and contrast_type prepended to identify the rank and contrast criterion used.

See Also

linkage_distances, linkage_turnover

Other Objective–decision linkage: linkage_distances(), linkage_transition(), linkage_turnover()

Examples

```
# Load a complete simulated multi-action problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(
    alias = "cost",
    include_pu_cost = FALSE
  ) |>
  add_objective_max_effect(
```

```
    alias = "benefit"
  ) |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 5),
    normalize_weights = TRUE
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(problem, verbose = FALSE)
  solutions <- solve(problem)

  linkage <- linkage_distances(
    solutions,
    objectives = c("cost", "benefit")
  )

  linkage_contrasts(
    linkage,
    type = "objective_similar",
    n = 3
  )

  turnover <- linkage_turnover(
    solutions,
    objectives = c("cost", "benefit")
  )

  linkage_contrasts(
    turnover,
    type = "high_reconfiguration",
    n = 3
  )
}
```

`linkage_distances` *Compare objective and decision distances*

Description

Measure separately how far pairs of stored solutions are in objective space and how much their planning-unit/action assignments differ.

Usage

```
linkage_distances(
  x,
  objectives = NULL,
  pairs = NULL,
```

```
  objective_metric = c("euclidean", "manhattan", "chebyshev"),
  decision_metric = c("jaccard", "hamming")
)
```

Arguments

x	A solutionset-class object returned by solve.
objectives	Optional character vector with two or more unique objective aliases. If NULL, all registered objectives are used.
pairs	Either NULL, or a non-empty data.frame containing numeric positive-integer from_solution and to_solution columns. Output from frontier_neighbors can be supplied directly.
objective_metric	Objective-space distance metric: "euclidean", "manhattan", or "chebyshev".
decision_metric	Decision-space metric: "jaccard" or "hamming".

Details

Objective values are oriented to minimization and normalized using the solutions retained in the supplied SolutionSet. Decision distances are always calculated on the complete planning-unit/action assignment space represented by the supplied solutions.

If pairs = NULL, all unordered pairs are generated and oriented from worse to better on the first selected objective. If pairs is supplied, its from_solution and to_solution direction is preserved. Distances are symmetric, but signed objective changes and action additions or removals depend on pair direction.

No combined linkage score is calculated.

Value

A data.frame with one row per directed pair, including objective_distance, decision_similarity, decision_distance, decision-change counts, and objective-specific from/to values, changes, and improvements.

See Also

frontier_neighbors, frontier_distances, selection_similarity, linkage_transition

Other Objective–decision linkage: linkage_contrasts(), linkage_transition(), linkage_turnover()

Examples

```
# Load a complete simulated multi-action problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
```

```
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(
    alias = "cost",
    include_pu_cost = FALSE
  ) |>
  add_objective_max_effect(
    alias = "benefit"
  ) |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 5),
    normalize_weights = TRUE
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(problem, verbose = FALSE)
  solutions <- solve(problem)

  linkage <- linkage_distances(
    solutions,
    objectives = c("cost", "benefit")
  )

  head(linkage)

  neighbors <- frontier_neighbors(
    solutions,
    objectives = c("cost", "benefit")
  )

  linkage_distances(
    solutions,
    objectives = c("cost", "benefit"),
    pairs = neighbors
  )
}
```

`linkage_transition` *Describe the transition between two solutions*

Description

Explain the objective and spatial changes required to transform one stored solution into another.

Usage

```
linkage_transition(x, from, to, objectives = NULL)
```

Arguments

x	A solutionset-class object returned by solve.
from	A single numeric positive-integer id identifying the starting stored solution.
to	A single numeric positive-integer id identifying the destination stored solution.
objectives	Optional character vector with one or more objective aliases. If NULL, all registered objectives are reported.

Details

The transition is directional. Reversing from and to reverses signed objective changes, additions and removals, and planning-unit activations and deactivations. Canonical planning-unit states are obtained from get_solution_states.

The complete landscape is always returned in transitions, including planning units whose state is unchanged. To keep the action-level output compact, actions contains assignments selected in at least one of the two solutions; added, removed, and retained actions remain explicit.

Value

An object of class multiscape_linkage_transition containing:

- summary: the directed solution ids and transition counts;
- objectives: objective values and signed changes;
- transitions: one row per planning unit in the complete landscape, including unchanged units;
- actions: action assignments selected in either solution;
- state_matrix: counts of planning-unit state transitions.

See Also

get_solution_states, linkage_distances, frontier_neighbors

Other Objective–decision linkage: linkage_contrasts(), linkage_distances(), linkage_turnover()

Examples

```
# Load a complete simulated multi-action problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
```

```
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(
    alias = "cost",
    include_pu_cost = FALSE
  ) |>
  add_objective_max_effect(
    alias = "benefit"
  ) |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 5),
    normalize_weights = TRUE
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(problem, verbose = FALSE)
  solutions <- solve(problem)

  neighbors <- frontier_neighbors(
    solutions,
    objectives = c("cost", "benefit")
  )

  transition <- linkage_transition(
    solutions,
    from = neighbors$from_solution[1],
    to = neighbors$to_solution[1],
    objectives = c("cost", "benefit")
  )

  transition$summary
  transition$objectives

  transition$transitions[
    transition$transitions$changed,
    ,
    drop = FALSE
  ]
}
```

linkage_turnover *Measure decision turnover along an objective-space neighborhood*

Description

Measure decision-space change between selected solution pairs and relate it to their normalized objective-space distance.

Usage

```
linkage_turnover(
  x,
  objectives = NULL,
  pairs = NULL,
  objective_metric = c("euclidean", "manhattan", "chebyshev"),
  decision_metric = c("jaccard", "hamming"),
  tolerance = sqrt(.Machine$double.eps)
)
```

Arguments

x	A solutionset-class object returned by solve.
objectives	Optional character vector with two or more unique objective aliases. If NULL, objectives stored on pairs are reused when available; otherwise all registered objectives are used.
pairs	Either NULL, or a non-empty data.frame containing numeric positive-integer from_solution and to_solution columns. Output from frontier_neighbors can be supplied directly.
objective_metric	Objective-space metric: "euclidean", "manhattan", or "chebyshev". The metric is also used to generate neighbors when pairs = NULL.
decision_metric	Decision-space metric over the complete planning-unit/action assignment space: "jaccard" or "hamming".
tolerance	A single finite non-negative number. Pairs whose normalized objective distance is less than or equal to this value are treated as objective ties.

Details

Decision turnover is the decision_distance returned by linkage_distances. The reconfiguration rate relates that turnover to normalized objective-space separation:

$$R_{rs} = d_X(r, s)/d_Z(r, s).$$

It is a unitless ratio, not a temporal rate or a percentage of landscape area. No additional turnover column is returned because it would duplicate decision_distance.

A pair is an objective tie when `objective_distance <= tolerance`. Tied pairs receive `NA_real_` for `reconfiguration_rate`, including pairs with identical decisions, avoiding undefined or infinite ratios. Non-tied pairs with identical decisions receive a rate of zero.

If `pairs = NULL`, neighboring pairs are generated with `frontier_neighbors` using `method = "auto"` and the requested `objective_metric`. Supplied pairs preserve their direction. Reconfiguration rates should be compared only among analyses using the same objectives, supplied `SolutionSet`, normalization basis, and distance metrics.

Value

A `data.frame` extending the output of `linkage_distances` with logical `objective_tie` and numeric `reconfiguration_rate` columns. Decision turnover is the existing `decision_distance` column.

See Also

`frontier_neighbors`, `linkage_distances`, `linkage_contrasts`

Other Objective–decision linkage: `linkage_contrasts()`, `linkage_distances()`, `linkage_transition()`

Examples

```
# Load a complete simulated multi-action problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(
    alias = "cost",
    include_pu_cost = FALSE
  ) |>
  add_objective_max_effect(
    alias = "benefit"
  ) |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 5),
    normalize_weights = TRUE
  )
```

```
if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(problem, verbose = FALSE)
  solutions <- solve(problem)

  neighbors <- frontier_neighbors(
    solutions,
    objectives = c("cost", "benefit")
  )

  turnover <- linkage_turnover(
    solutions,
    objectives = c("cost", "benefit"),
    pairs = neighbors
  )

  turnover

  turnover[
    turnover$objective_tie &
      turnover$decision_distance > 0,
    ,
    drop = FALSE
  ]
}
```

`load_ecosystem_services` *Load the ecosystem-services spatial example*

Description

Load the paired planning units and feature raster used by the integrated planning vignette. This is a different landscape from both the compact simulated example and Meseta. The vignette derives its feature catalogue and reference amounts from the raster.

Usage

```
load_ecosystem_services()
```

Value

A named list containing:

planning_units An `sf` object with 30,496 planning units, including cost, area and historical lock columns.

feature_raster A `terra::SpatRaster` with four ecosystem-service layers.

Examples

```
services <- load_ecosystem_services()
nrow(services$planning_units)
names(services$feature_raster)
```

`load_meseta` *Load the Meseta Iberica spatial action example*

Description

Load the complete inputs used by the introductory README tutorial. These earlier available data support an adaptation of the Meseta case study, not an exact reproduction of the final 2025 article.

Usage

```
load_meseta()
```

Details

The bundled inputs are redistributed under GPL-3.0. Their source and preparation notes are recorded in `provenance`.

The action identifiers in `actions$id`, `action_costs$action` and `outcomes$action` consistently use `Afforestation`, `BAU`, `FarmReturn` and `Firesmart`. The tables can be passed directly to `add_actions()` and `add_effects()`, without recoding identifiers. Scripts that explicitly select numeric action identifiers must use these names instead. Original identifiers remain available in `provenance$action_id_lookup`. All amounts, costs, targets, reference values and geometries are preserved. Planning-unit costs are retained; use `include_pu_cost = FALSE` in `add_objective_min_cost()` to count only action implementation costs. Maps are derived when loading, without storing additional copies of geometry in package data. Map columns use the names in `actions$name`; rows follow `planning_units` order, and row names contain planning-unit identifiers.

The five feature maps illustrate contrasting action scenarios (three species and two ecosystem-service features), not ecological importance. Missing outcome rows are displayed as zero, following the dataset convention. These maps describe inputs under each action, not optimized selections.

Value

A named list containing:

planning_units An `sf` object with 11,109 planning units.

features A catalogue of 155 features.

dist_features The explicit comparison reference used in the tutorial.

actions A catalogue of four management strategies, with readable character identifiers in `id` (also provided in `name`).

action_costs Costs by planning unit and action.

outcomes Outcomes by planning unit, action and feature.

targets Representation targets for all features.

boundary The supplied weighted spatial relation. Replacing it with a newly derived boundary can change the optimization problem.

provenance Data sources and preparation notes. `action_id_lookup` records the original numeric `source_id` and the returned character `id` for each action.

maps Ready-to-plot `sf` views: `costs`, `DENMINO`, `CISJUNC`, `MLE`, `AGRICULTURE` and `ROS`. Each view contains four strategy columns and geometry, so `plot()` displays the four action scenarios directly.

Examples

```
meseta <- load_meseta()
names(meseta)
meseta$actions
plot(meseta$maps$costs)
plot(meseta$maps$DENMINO)
```

---

load_sim_features_raster

Example feature raster

Description

Load the example feature raster shipped with the package. This access is retained for compatibility; use `load_ecosystem_services()` to load its matching planning units and raster together.

Usage

```
load_sim_features_raster()
```

Value

A `terra::SpatRaster`.

---

load_sim_multiaction

Load the simulated spatial multi-action example

Description

Load a compact, deterministic dataset for examples of multi-objective spatial planning. The landscape contains 64 square planning units, two spatially structured features, and two mutually exclusive candidate actions.

Usage

```
load_sim_multiaction()
```

Value

A named list containing:

`planning_units` An `sf` object with 64 planning units.

`features` A feature catalogue.

`dist_features` Feature amounts by planning unit.

`actions` The action catalogue.

`action_costs` Action costs by planning unit and action.

`effect_assumptions` Relative-change assumptions by action and feature.

`effects` Explicit signed changes by planning unit, action, and feature.

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()
names(example_data)
utils::head(example_data$planning_units)
```

---

mo_control *Control multi-objective run behavior*

Description

[Deprecated]

`mo_control()` has been replaced by `set_runs_control`.

The argument `slack_upper_bound` is deprecated and ignored. For AUGMECON, use the `slack_upper_bound` argument directly in `set_method_augmecon`.

Usage

```
mo_control(
  stop_on_infeasible = FALSE,
  stop_on_no_solution = FALSE,
  stop_on_error = TRUE,
  slack_upper_bound = NULL
)
```

Arguments

`stop_on_infeasible`
: Logical. If `TRUE`, stop the complete multi-objective workflow when a run is reported as infeasible.

`stop_on_no_solution`
: Logical. If `TRUE`, stop when a run does not return a usable solution vector.

`stop_on_error` Logical. If `TRUE`, stop on unexpected errors.

`slack_upper_bound`
Deprecated. Use `slack_upper_bound` in `set_method_augmecon` instead.

Value

An object of class `RunsControl`, `MOControl`, and `list`.

See Also

`set_runs_control`, `set_method_augmecon`

`plot_spatial_actions` *Plot selected actions in space*

Description

Plot the spatial distribution of selected actions from a `solutionset-class` object returned by `solve`.

This function maps the selected planning unit–action pairs returned by `get_actions` onto the planning-unit geometry stored in the associated `Problem` object.

Usage

```
plot_spatial_actions(
  x,
  solutions = NULL,
  actions = NULL,
  layout = NULL,
  max_facets = 4L,
  nrow = NULL,
  ncol = NULL,
  ...,
  base_alpha = 0.08,
  selected_alpha = 0.95,
  base_fill = "grey95",
  base_color = NA,
  selected_color = NA,
  draw_borders = FALSE,
  show_base = TRUE,
  fill_values = NULL,
  fill_na = "grey80",
  use_viridis = TRUE
)
```

Arguments

x	A solutionset-class object returned by solve.
solutions	Optional integer vector of solution ids. If NULL, the first available solution is plotted by default.
actions	Optional action subset to display. Entries may match action ids or action-set labels.
layout	Character string controlling the layout. Must be one of "single" or "facet". If NULL, the default is "single".
max_facets	Maximum number of action facets shown when actions is NULL and faceting would otherwise create many panels.
nrow, ncol	Optional numbers of rows and columns used to arrange facets.
...	Reserved for future extensions.
base_alpha	Numeric value in $[0, 1]$ giving the alpha of the base planning-unit layer.
selected_alpha	Numeric value in $[0, 1]$ giving the alpha of the highlighted action layer.
base_fill	Fill colour for the base planning-unit layer.
base_color	Border colour for the base planning-unit layer.
selected_color	Border colour for highlighted layers.
draw_borders	Logical. If FALSE, borders are not drawn.
show_base	Logical. If TRUE, draw the base planning-unit layer underneath the highlighted output.
fill_values	Optional named vector of colours for discrete action maps.
fill_na	Fill colour for missing values.
use_viridis	Logical. If TRUE and the **viridis** package is available, use viridis discrete scales.

Details

Let $x_{ia} \in \{0, 1\}$ denote whether action a is selected in planning unit i. This function plots the selected (pu, action) pairs in geographic space.

If layout = "facet" and only one run is plotted, one panel is drawn per action.

If layout = "single", all selected actions are drawn in a single map using discrete fills. If more than one action is selected in the same planning unit, the action labels are collapsed using "+".

When plotting multiple runs, only layout = "single" is supported.

Planning-unit geometry must be available in the associated problem object.

Value

Invisibly returns a ggplot object.

See Also

get_actions, plot_spatial_planning_units, plot_spatial_features

Examples

```
if (
  requireNamespace("sf", quietly = TRUE) &&
  requireNamespace("ggplot2", quietly = TRUE) &&
  requireNamespace("rcbc", quietly = TRUE)
) {
  # Load a complete simulated planning problem.
  example_data <- load_sim_multiaction()

  problem <- create_problem(
    pu = example_data$planning_units,
    features = example_data$features,
    dist_features = example_data$dist_features,
    cost = "cost"
  ) |>
    add_actions(
      example_data$actions,
      cost = example_data$action_costs
    ) |>
    add_effects(
      example_data$effect_assumptions
    ) |>
    add_constraint_targets_relative(0.05) |>
    add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
    add_objective_max_effect(alias = "benefit") |>
    set_method_weighted_sum(
      aliases = c("cost", "benefit"),
      runs = set_runs_grid(n = 3),
      normalize_weights = TRUE
    ) |>
    set_solver_cbc(verbose = FALSE)

  solutions <- solve(problem)

  plot_spatial_actions(
    solutions,
    layout = "single"
  )
}
```

---

plot_spatial_features *Plot spatial feature values from a solution set*

Description

Plot feature values in space from a `solutionset-class` object returned by `solve`.

This function combines baseline feature amounts from the associated `Problem` object with positive effects induced by the actions selected in each stored run to produce planning-unit-level feature maps. Selected actions are obtained through `get_actions`.

Usage

```
plot_spatial_features(
  x,
  solutions = NULL,
  features = NULL,
  value = c("final", "baseline", "benefit"),
  layout = NULL,
  max_facets = 4L,
  ...,
  base_alpha = 0.1,
  selected_alpha = 0.9,
  base_fill = "grey92",
  base_color = NA,
  selected_color = NA,
  draw_borders = FALSE,
  show_base = TRUE,
  fill_na = "grey80",
  use_viridis = TRUE
)
```

Arguments

x	A solutionset-class object returned by solve.
solutions	Optional integer vector of solution ids. If NULL, the first available solution is plotted by default.
features	Optional feature subset to display. Matching is attempted against both feature ids and feature names.
value	Character string indicating which feature quantity to plot. Must be one of "final", "baseline", or "benefit".
layout	Character string controlling the layout. Must be one of "single" or "facet". If NULL, the default is "facet".
max_facets	Maximum number of feature facets shown when features = NULL and faceting would otherwise create many panels.
...	Reserved for future extensions.
base_alpha	Unused in the current feature view, kept for interface consistency.
selected_alpha	Unused in the current feature view, kept for interface consistency.
base_fill	Unused in the current feature view, kept for interface consistency.
base_color	Unused in the current feature view, kept for interface consistency.
selected_color	Border colour for filled feature polygons.
draw_borders	Logical. If FALSE, borders are not drawn.
show_base	Unused in the current feature view, kept for interface consistency.
fill_na	Fill colour for missing values.
use_viridis	Logical. If TRUE and the **viridis** package is available, use a continuous viridis scale.

Details

For each planning unit i and feature f, the plotted quantities are:

$$\text{baseline}_{if},$$

$$\text{benefit}_{if},$$

$$\text{final}_{if} = \text{baseline}_{if} + \text{benefit}_{if}.$$

In the current implementation:

- `baseline` is the summed baseline amount from `dist_features`;
- `benefit` is the summed positive effect from selected actions;
- `final` is `baseline` + `benefit`.

Negative effects are not subtracted in this plotting method. Therefore, `value = "final"` should be interpreted as baseline plus selected positive effects under the current plotting logic.

If `layout = "facet"` and only one run is plotted, one panel is drawn per feature.

If multiple runs are plotted, exactly one feature must be requested, and faceting is done by run.

Planning-unit geometry must be available in the associated problem object.

Value

Invisibly returns a `ggplot` object.

See Also

`get_features`, `plot_spatial_planning_units`, `plot_spatial_actions`

Examples

```
if (
  requireNamespace("sf", quietly = TRUE) &&
  requireNamespace("ggplot2", quietly = TRUE) &&
  requireNamespace("rcbc", quietly = TRUE)
) {
  # Load a complete simulated planning problem.
  example_data <- load_sim_multiaction()

  problem <- create_problem(
    pu = example_data$planning_units,
    features = example_data$features,
    dist_features = example_data$dist_features,
    cost = "cost"
  ) |>
    add_actions(
      example_data$actions,
      cost = example_data$action_costs
    ) |>
    add_effects(
      example_data$effect_assumptions
```

```
    ) |>
    add_constraint_targets_relative(0.05) |>
    add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
    add_objective_max_effect(alias = "benefit") |>
    set_method_weighted_sum(
      aliases = c("cost", "benefit"),
      runs = set_runs_grid(n = 3),
      normalize_weights = TRUE
    ) |>
    set_solver_cbc(verbose = FALSE)

  solutions <- solve(problem)

  plot_spatial_features(
    solutions,
    features = 1,
    value = "final"
  )
}
```

---

plot_spatial_planning_units

Plot selected planning units in space

Description

Plot the spatial distribution of selected planning units from a solutionset-class object returned by solve.

This function maps the planning-unit selection summary returned by get_planning_units onto the planning-unit geometry stored in the associated Problem object.

Usage

```
plot_spatial_planning_units(
  x,
  solutions = NULL,
  ...,
  base_alpha = 0.1,
  selected_alpha = 0.9,
  base_fill = "grey92",
  base_color = NA,
  selected_color = NA,
  draw_borders = FALSE,
  show_base = TRUE
)
```

Arguments

x	A solutionset-class object returned by solve.
solutions	Optional integer vector of solution ids. If NULL, the first available solution is plotted by default.
...	Reserved for future extensions.
base_alpha	Numeric value in $[0, 1]$ giving the alpha of the base planning-unit layer.
selected_alpha	Numeric value in $[0, 1]$ giving the alpha of the selected planning-unit layer.
base_fill	Fill colour for the base planning-unit layer.
base_color	Border colour for the base planning-unit layer.
selected_color	Border colour for selected planning units.
draw_borders	Logical. If FALSE, borders are not drawn.
show_base	Logical. If TRUE, draw the base planning-unit layer underneath the selected units.

Details

Let $w_i \in \{0, 1\}$ denote the planning-unit selection variable for planning unit i. This function plots the user-facing selected == 1 representation of w_i.

If several runs are requested, the output is faceted by run_id.

Planning-unit geometry must be available in the associated problem object.

Value

Invisibly returns a ggplot object.

See Also

get_planning_units, plot_spatial_actions, plot_spatial_features, plot_spatial_planning_units

Examples

```
if (
  requireNamespace("sf", quietly = TRUE) &&
  requireNamespace("ggplot2", quietly = TRUE) &&
  requireNamespace("rcbc", quietly = TRUE)
) {
  # Load a complete simulated planning problem.
  example_data <- load_sim_multiaction()

  problem <- create_problem(
    pu = example_data$planning_units,
    features = example_data$features,
    dist_features = example_data$dist_features,
    cost = "cost"
  ) |>
    add_actions(
      example_data$actions,
```

```
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
  add_objective_max_effect(alias = "benefit") |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 3),
    normalize_weights = TRUE
  ) |>
  set_solver_cbc(verbose = FALSE)

solutions <- solve(problem)

plot_spatial_planning_units(solutions)
}
```

`plot_spatial_pu` *Plot selected planning units in space*

Description

[Deprecated]

`plot_spatial_pu()` has been replaced by `plot_spatial_planning_units`.

Usage

```
plot_spatial_pu(
  x,
  solutions = NULL,
  ...,
  base_alpha = 0.1,
  selected_alpha = 0.9,
  base_fill = "grey92",
  base_color = NA,
  selected_color = NA,
  draw_borders = FALSE,
  show_base = TRUE
)
```

Arguments

`x`	A `solutionset-class` object returned by `solve`.
`solutions`	Optional integer vector of solution ids. If `NULL`, the first available solution is plotted by default.

...	Reserved for future extensions.
base_alpha	Numeric value in $[0, 1]$ giving the alpha of the base planning-unit layer.
selected_alpha	Numeric value in $[0, 1]$ giving the alpha of the selected planning-unit layer.
base_fill	Fill colour for the base planning-unit layer.
base_color	Border colour for the base planning-unit layer.
selected_color	Border colour for selected planning units.
draw_borders	Logical. If FALSE, borders are not drawn.
show_base	Logical. If TRUE, draw the base planning-unit layer underneath the selected units.

Value

Invisibly returns a ggplot object.

See Also

plot_spatial_planning_units

---

plot_tradeoff *Plot trade-offs from a solution set*

Description

Plot pairwise trade-offs among objective values stored in a solutionset-class object.

This function is intended for workflows in which the solution set contains one row per run and two or more objective-value columns of the form value_*.

If exactly two objectives are selected, the function returns a single scatterplot. If three or more objectives are selected, all pairwise combinations are plotted using facets.

Usage

```
plot_tradeoff(
  x,
  objectives = NULL,
  color_by = NULL,
  all_pairs = NULL,
  connect = FALSE,
  label_runs = FALSE,
  point_size = 3,
  line_alpha = 0.5,
  text_size = 3,
  ...
)
```

Arguments

x	A solutionset-class object.
objectives	Optional character vector of objective aliases to display. These must match the suffixes of the value_* columns in x$solution$runs. If NULL, all available objective columns are used.
color_by	Optional character scalar used to colour points. This may be either one of the selected objective aliases or one of the run-level columns "run_id", "status", "runtime", or "gap".
all_pairs	Logical. If TRUE, allow plotting all pairwise combinations even when more than four objectives are selected. If NULL, it is treated as FALSE.
connect	Logical. If TRUE, connect points by run order within each panel.
label_runs	Logical. If TRUE, add run labels to points.
point_size	Numeric point size.
line_alpha	Numeric alpha value for connecting lines.
text_size	Numeric size for run labels.
...	Reserved for future extensions.

Details

This function reads the run-level table stored in x$solution$runs. It expects objective values to be stored in columns whose names begin with "value_".

If the available objective columns are, for example, value_cost, value_benefit, and value_frag, then the corresponding objective aliases are "cost", "benefit", and "frag".

Let $f_k(r)$ denote the value of objective k in run r. This function visualizes pairwise projections of the run table of the form:

$$(f_k(r), f_\ell(r))$$

for selected pairs of objectives k, ℓ.

If exactly two objectives are selected, a single panel is produced.

If three or more objectives are selected, all pairwise combinations are generated:

$$\{(k, \ell) : k < \ell,\ k, \ell \in \mathcal{O}\},$$

where $\mathcal{O}$ is the selected set of objective aliases.

By default, plotting more than four objectives is not allowed unless all_pairs = TRUE, because the number of panels grows quadratically in the number of objectives.

Colouring

If color_by is supplied, points are coloured by either:

- one of the selected objective aliases, in which case the corresponding value_* column is used;
- or one of the run-level columns run_id, status, runtime, or gap.

Connecting runs

If `connect = TRUE`, runs are connected in their current table order within each panel. This can be useful when runs correspond to an ordered scan of weights, ϵ-levels, or frontier points, but it should be used with care when run order has no substantive meaning.

Run labels

If `label_runs = TRUE`, each point is labelled by its `run_id`. If the **ggrepel** package is available, repelled labels are used.

Value

Invisibly returns a `ggplot` object.

See Also

`solve`, `solutionset-class`

Examples

```
if (
  requireNamespace("ggplot2", quietly = TRUE) &&
  requireNamespace("rcbc", quietly = TRUE)
) {
  # Load a complete simulated planning problem.
  example_data <- load_sim_multiaction()

  problem <- create_problem(
    pu = example_data$planning_units,
    features = example_data$features,
    dist_features = example_data$dist_features,
    cost = "cost"
  ) |>
    add_actions(
      example_data$actions,
      cost = example_data$action_costs
    ) |>
    add_effects(
      example_data$effect_assumptions
    ) |>
    add_constraint_targets_relative(0.05) |>
    add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
    add_objective_max_effect(alias = "benefit") |>
    set_method_weighted_sum(
      aliases = c("cost", "benefit"),
      runs = set_runs_grid(n = 3),
      normalize_weights = TRUE
    ) |>
    set_solver_cbc(verbose = FALSE)

  solutions <- solve(problem)

  plot_tradeoff(
```

```
    solutions,
    objectives = c("cost", "benefit")
  )
}
```

`problem-class` *Problem class*

Description

The `Problem` class is the central container used by **multiscape** to represent a planning problem before, during, and after model construction.

A `Problem` object stores the full problem specification in a modular way. This includes the baseline planning data, optional spatial metadata, action definitions, effects, profit, targets, constraints, objective registrations, solver settings, and, when available, a built optimization model or model snapshot.

In other words, `Problem` is the persistent object that connects the full `multiscape` workflow:

```
create_problem()
-> add_*() / set_*()
-> solve()
```

Details

Repeated calls

Action catalogues, effects, profit, solver configuration, multi-objective methods, and unaliased single objectives can each be defined once. A second definition raises an error. Action sets, named spatial relations, aliased objectives, and distinct constraints accumulate; duplicate definitions are rejected. Locks accumulate compatible states, allow identical repeats, and reject contradictory states. To compare scenarios or configurations, derive each alternative from a common problem before the relevant definition.

Conceptual role

The `Problem` class is designed for a data-first and modular workflow. User-facing functions do not usually modify a solver object directly. Instead, they enrich the `Problem` object by storing new specifications in its internal `data` field.

Thus, a `Problem` object should be understood as a structured container for the mathematical planning problem, not necessarily as a built optimization model.

Before `solve` is called, the object may contain only input data and user specifications. During or after solving, it may additionally contain a built model pointer, model metadata, and solver-related information.

Core mathematical interpretation

At a high level, the `Problem` object stores the ingredients required to define an optimization problem over:

- planning units $i \in \mathcal{P}$,
- features $f \in \mathcal{F}$,
- actions $a \in \mathcal{A}$,
- optional spatial relations over planning units,
- and user-defined objectives and constraints.

The baseline ecological state is typically stored through a planning unit–feature table of amounts:

$$a_{if} \geq 0,$$

where a_{if} is the baseline amount of feature f in planning unit i.

Subsequent functions then add action feasibility, effects, profit, targets, spatial relations, and optimization settings to this baseline representation.

How objects are created

`Problem` objects are usually created by `create_problem`.

After creation, downstream functions such as `add_actions`, `add_effects`, `add_profit`, `add_constraint_targets_absolu` `add_constraint_targets_relative`, spatial relation constructors, objective setters, and solver setters extend the internal `data` list.

Internal storage

The class contains a single field:

`data` A named `list` storing the full problem specification, metadata, and, when available, built-model information.

Common entries of `data` include:

`pu` Planning-unit table.

`features` Feature table.

`actions` Action catalog.

`action_sets` Named combinations of actions, stored as a long table of set–action memberships. These definitions do not impose joint selection or change the optimization model.

`dist_features` Planning unit–feature baseline amounts.

`dist_actions` Feasible planning unit–action pairs.

`dist_effects` Action effects by planning unit, action, and feature.

`dist_profit` Profit by planning unit–action pair.

`pu_sf` Planning-unit geometry when available.

`pu_coords` Planning-unit coordinates when available.

`spatial_relations` Registered spatial relations.

`targets` Stored target specifications.

`constraints` Stored user-defined constraints, including per-unit action counts in `constraints$action_cardinality` and logical relations in `constraints$action_relations`.

`effects_original`, `joint_effects`, `effect_terms` Canonical supplied totals, supplied joint totals, and their signed subset corrections, when registered sets occur in modern effect input. Missing interactions are assumed zero as recorded in `effects_meta`.

`objectives` Registered atomic objectives for single- or multi-objective workflows.

`method` Stored multi-objective method configuration, when applicable.

`solve_args` Stored solver settings.

`model_ptr` Pointer to a built optimization model, when available.

`model_args` Metadata describing model construction.

`model_list` Optional exported model snapshot or representation.

`meta` Auxiliary metadata, including model-dirty flags and other bookkeeping fields.

Not every `Problem` object contains all of these entries. The content of `data` depends on how far the workflow has progressed.

Lifecycle

A `Problem` object typically moves through the following stages:

1. input stage: baseline planning units, features, and feature distributions are stored,
2. specification stage: actions, effects, targets, objectives, constraints, and spatial relations are added,
3. model stage: an optimization model is built from the stored specification,
4. solve stage: the model is solved and results are returned in a separate `Solution` or `SolutionSet` object.

The `Problem` object itself is not the solution. It is the structured problem definition from which a solution can be obtained.

Value

No return value. This page documents the `Problem` class.

Methods

`print()` Print a structured summary of the stored problem specification, including data, actions and effects, spatial inputs, targets and constraints, and model status. If a model has already been built, additional dimensions and auxiliary-variable information are displayed.

`show()` Alias of `print()`.

`repr()` Return a short one-line representation of the object.

`getData(name)` Return a named entry from `self$data`.

`getPlanningUnitsAmount()` Return the number of planning units stored in `x$data$pu`.

`getMonitoringCosts()` Return the planning-unit cost vector, typically taken from `x$data$pu$cost`.

`getFeatureAmount()` Return the number of stored features.

`getFeatureNames()` Return feature names from `x$data$features$name`, or feature ids if names are unavailable.

`getActionCosts()` Return action-level costs from `x$data$dist_actions$cost` when available.

`getActionsAmount()` Return the number of stored actions.

Printing and diagnostics

The `print()` method is intended as a quick diagnostic summary. It helps users understand:

- what data have already been loaded,
- whether actions, effects, spatial relations, targets, and constraints have been registered,
- whether objectives and methods have been configured,
- whether a model has already been built,
- and whether the object appears ready to be solved.

In particular, the model section of the printed output summarizes whether the current problem specification is incomplete, ready, or already materialized as a built optimization model.

See Also

`create_problem`, `add_actions`, `add_action_sets`, `add_constraint_action_cardinality`, `add_effects`, `add_constraint_targets_absolute`, `solve`

---

run_grid *Define an automatic multi-objective run grid*

Description

[Deprecated]

`run_grid()` has been replaced by `set_runs_grid`.

The argument `include_extremes` is deprecated and ignored. Boundary levels are now always included.

Usage

```
run_grid(n, include_extremes = TRUE)
```

Arguments

`n`	Integer. Resolution of the automatically generated run design. Must be at least 2.
`include_extremes`	Deprecated. Boundary levels are now always included.

Value

An object of class `RunGrid` and `RunDesign`.

See Also

`set_runs_grid`

run_manual *Define a manual multi-objective run design*

Description

[Deprecated]

`run_manual()` has been replaced by `set_runs_manual`.

Usage

```
run_manual(x)
```

Arguments

x	A non-empty `data.frame` with one row per optimization run. Run-design columns must be named using the `weight_<alias>` or `eps_<alias>` convention.

Value

An object of class `RunManual` and `RunDesign`.

See Also

`set_runs_manual`

selection_consistency *Summarize consistency of planning-unit states*

Description

Summarize recurrence and disagreement of canonical planning-unit states across stored solutions.

Usage

```
selection_consistency(x, solution_groups = NULL)
```

Arguments

x	A `solutionset-class` object returned by `solve`.
solution_groups	Optional named list assigning every stored numeric solution id to exactly one group. If `NULL`, all solutions form the group `"all"`.

Details

States are obtained from get_solution_states. All stored solutions contribute equally, and all planning units are returned. solution_groups can partition the stored solutions into named groups; consistency is then summarized separately within each group.

Frequencies describe the proportion of stored solutions in the corresponding group. They are conditional summaries of the supplied SolutionSet, not probabilities or formal measures of uncertainty. variable is TRUE when more than one state occurs for a planning unit within a solution group.

Value

A data.frame with one row per planning unit and solution group, including selection, managed and unmanaged frequencies, dominant state and frequency, number of observed states, whether the unit is variable within the group, Shannon entropy, and normalized entropy.

See Also

get_solution_states, selection_frequency, selection_similarity, solution_filter

Examples

```
# Load a complete simulated multi-action problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(
    alias = "cost",
    include_pu_cost = FALSE
  ) |>
  add_objective_max_effect(
    alias = "benefit"
  ) |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 5),
    normalize_weights = TRUE
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
```

```
  problem <- set_solver_cbc(problem, verbose = FALSE)
  solutions <- solve(problem)

  consistency <- selection_consistency(solutions)
  head(consistency)

  solution_ids <- get_runs(solutions)$solution_id
  solution_ids <- solution_ids[!is.na(solution_ids)]

  if (length(solution_ids) >= 2L) {
    cut <- ceiling(length(solution_ids) / 2)
    groups <- list(
      first = solution_ids[seq_len(cut)],
      second = solution_ids[seq.int(cut + 1L, length(solution_ids))]
    )

    selection_consistency(
      solutions,
      solution_groups = groups
    )
  }
}
```

`selection_frequency` *Calculate selection frequency across solutions*

Description

Calculate how frequently each planning-unit/action assignment is selected across the stored solutions in a `solutionset-class` object.

Usage

```
selection_frequency(x)
```

Arguments

x	A `solutionset-class` object returned by `solve`.

Details

Selection frequency is calculated at the planning-unit/action level. This is the canonical decision representation used by this function because it preserves differences between solutions that select the same planning unit but assign different actions.

For each planning-unit/action pair, the frequency is:

$$f_{ia} = \frac{\sum_{s \in S} x_{ias}}{|S|},$$

where x_{ias} equals one when planning unit i receives action a in solution s, and zero otherwise.

The result is computed over all stored solutions in the supplied `SolutionSet`. To calculate frequencies for only a subset of solutions, first use `solution_filter` or `solution_unique`.

For simple conservation-planning problems without explicit actions, selected planning units are represented using the canonical action name `"conservation"`.

Selection frequency measures recurrence across the supplied solutions. It should not automatically be interpreted as formal irreplaceability because it depends on the solutions included, their sampling across objective space, and whether duplicate or dominated solutions have been retained.

Value

A `data.frame` with one row per planning-unit/action pair and the following columns:

- `pu`: planning-unit identifier;
- `action`: action identifier or name;
- `n_selected`: number of stored solutions in which the planning-unit/action pair is selected;
- `n_solutions`: total number of stored solutions considered;
- `frequency`: proportion of stored solutions in which the pair is selected.

See Also

`selection_similarity`, `selection_consistency`, `solution_filter`, `solution_unique`, `get_actions`, `get_planning_units`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
  add_objective_max_effect(alias = "benefit") |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 3),
    normalize_weights = TRUE
```

```
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(
    problem,
    verbose = FALSE
  )

  solutions <- solve(problem)

  # Frequency across all stored solutions
  frequency <- selection_frequency(solutions)
  frequency

  # Restrict the analysis to non-dominated solutions
  if (requireNamespace("moocore", quietly = TRUE)) {
    nondominated_solutions <- solution_filter(
      solutions,
      feasible_only = TRUE,
      nondominated = TRUE
    )

    selection_frequency(nondominated_solutions)
  }

  # Give each distinct decision configuration the same weight
  unique_solutions <- solution_unique(
    solutions,
    by = "decisions"
  )

  unique_frequency <- selection_frequency(
    unique_solutions
  )

  unique_frequency
}
```

`selection_similarity` *Calculate structural similarity among solutions*

Description

Calculate pairwise structural similarity among the stored solutions in a `solutionset-class` object.

Usage

```
selection_similarity(
  x,
```

```
  metric = c("jaccard", "hamming"),
  format = c("long", "matrix")
)
```

Arguments

x	A solutionset-class object returned by solve.
metric	Character. Similarity metric to use. One of "jaccard" or "hamming".
format	Character. Output format. If "long", return one row per pair of solutions. If "matrix", return a symmetric similarity matrix with solution ids as row and column names.

Details

Solutions are compared using their complete planning-unit/action assignment vectors. Consequently, two solutions that select the same planning unit but assign different actions are treated as structurally different.

For simple conservation-planning problems without explicit actions, selected planning units are represented using the canonical action name "conservation".

Two similarity metrics are supported:

- "jaccard" compares the sets of selected planning-unit/action assignments:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

  Jaccard similarity focuses on selected assignments and ignores joint absences. It is generally the preferred metric for sparse conservation and management portfolios.

- "hamming" calculates the proportion of decision-vector positions that are equal:

$$H(A, B) = \frac{1}{m} \sum_{k=1}^{m} I(A_k = B_k),$$

  where m is the number of feasible planning-unit/action assignments. Unlike Jaccard similarity, Hamming similarity includes shared non-selections.

For both metrics, similarity ranges from zero to one:

- 1 indicates identical assignment structures;
- 0 indicates no structural agreement under the selected metric.

The corresponding distance is calculated as:

$$D(A, B) = 1 - S(A, B).$$

The comparison is performed over all stored solutions in the supplied object. To compare only a subset, first use solution_filter or solution_unique.

Value

The two solution-id columns are returned as integers.

If `format = "long"`, a `data.frame` with columns:

- `solution_id_1`;
- `solution_id_2`;
- `similarity`;
- `distance`.

If `format = "matrix"`, a symmetric numeric matrix of similarities is returned. Its diagonal is equal to one.

The selected metric is stored in the `"metric"` attribute.

See Also

`selection_frequency`, `selection_consistency`, `solution_filter`, `solution_unique`, `get_actions`, `get_planning_units`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
  add_objective_max_effect(alias = "benefit") |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 3),
    normalize_weights = TRUE
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(
    problem,
    verbose = FALSE
  )
```

```
solutions <- solve(problem)

# Pairwise Jaccard similarity in long format
jaccard_long <- selection_similarity(
  solutions
)

jaccard_long

# Symmetric Jaccard similarity matrix
jaccard_matrix <- selection_similarity(
  solutions,
  format = "matrix"
)

jaccard_matrix

# Hamming similarity includes shared non-selections
hamming_long <- selection_similarity(
  solutions,
  metric = "hamming"
)

hamming_long

# Compare only structurally unique solutions
unique_solutions <- solution_unique(
  solutions,
  by = "decisions"
)

selection_similarity(
  unique_solutions,
  format = "matrix"
)
}
```

`set_method_augmecon` *Configure the augmented epsilon-constraint method (AUGMECON)*

Description

Explore trade-offs among registered objectives using AUGMECON. One objective is optimised directly, while the others are constrained by epsilon levels and contribute small, normalised improvement terms to discourage weakly efficient solutions.

Usage

```
set_method_augmecon(
  x,
  primary,
  aliases = NULL,
  runs = NULL,
  grid = NULL,
  n_points = NULL,
  include_extremes = NULL,
  lexicographic = TRUE,
  lexicographic_tol = 1e-09,
  augmentation = 0.001,
  slack_upper_bound = 1e+06,
  control = NULL
)
```

Arguments

x	A Problem with at least two registered objective aliases.
primary	Character scalar naming the objective optimised directly.
aliases	Optional character vector of registered objective aliases. By default, all registered objectives are included; primary must be one of them.
runs	A run design from set_runs_grid() or set_runs_manual(). Manual designs contain eps_<alias> columns for all secondary objectives.
grid, n_points, include_extremes	Deprecated run-design arguments; use runs instead. include_extremes is ignored because automatic grids always include their endpoints.
lexicographic	Logical. Whether to use lexicographic anchoring when computing automatic-grid extreme points. Defaults to TRUE.
lexicographic_tol	Non-negative tolerance for the lexicographic extreme-point calculation, in the objective's original units.
augmentation	Positive coefficient for the normalised slack reward. Defaults to 1e-3.
slack_upper_bound	Positive upper bound for AUGMECON slack variables. Defaults to 1e6; it must not exclude feasible values.
control	Optional run-failure settings from set_runs_control().

Details

Register at least two objectives with distinct alias values. The primary objective is optimised directly; the remaining objectives selected by aliases become epsilon-constrained objectives. For example, with woodland improvement as the primary objective and cost as secondary, each run seeks the greatest improvement while keeping cost below a ceiling:

$$\max\ f_{\text{woodland}}(x), \qquad f_{\text{cost}}(x) \leq \varepsilon_{\text{cost}}.$$

What augmentation changes. Ordinary epsilon-constraint may return a weakly efficient plan when another objective could improve without changing the primary objective. AUGMECON introduces a non-negative slack for every secondary constraint and rewards normalised slack through a small positive coefficient `augmentation` (Mavrotas, 2009). For a minimised cost objective, the slack measures how far actual cost lies *below* its epsilon ceiling. Schematically, with a maximised primary objective, AUGMECON uses

$$\max \; f_p(x) + \rho \sum_k \frac{s_k}{R_k},$$

where ρ is `augmentation`, s_k is the non-negative slack, and R_k is the payoff-table range of the corresponding secondary objective. The implementation handles the signs of minimisation and maximisation objectives internally. Choose a small augmentation so that secondary improvements do not unintentionally outweigh primary performance.

Automatic epsilon grids. Use `runs = set_runs_grid(n = 5)` to obtain five epsilon levels, including the endpoints, for each secondary objective. Levels and payoff-table ranges are calculated during `solve()`. With three objectives, for example, two secondary grids of five levels yield up to 25 combinations. When `lexicographic = TRUE`, extreme-point calculations use lexicographic anchoring, with `lexicographic_tol` specifying the tolerance.

Manual thresholds. Use `runs = set_runs_manual()` to supply explicit epsilon values, one row per run and one column named `eps_<alias>` for each secondary objective. For a minimised cost objective, `eps_cost = 20` means *cost at most 20*, not exactly 20. Unlike automatic grids, manual rows are not expanded into all possible combinations; use `expand.grid()` if needed.

Interpreting solutions. The augmentation reduces weak-efficiency problems, but a finite epsilon grid does not enumerate every Pareto-optimal plan, and different runs can produce identical spatial decisions. Numerical tolerances and finite solver time limits also matter. A numerical refinement pass may improve secondary slacks after an optimal augmented solve; solver settings and epsilon-bound checks are available in the solution diagnostics. `slack_upper_bound` must be large enough for valid slacks, without being unnecessarily loose. Use `set_runs_control()` to manage infeasible or unsuccessful runs.

Only one multi-objective method can be configured on a `Problem`; compare methods by starting from the same problem before calling `set_method_*()`. The older `grid`, `n_points`, and `include_extremes` arguments are deprecated in favour of `runs`.

Value

An updated `Problem` with the method stored in `x$data$method`. Payoff-table calculations and optimisation runs take place in `solve()`.

References

Mavrotas, G. (2009). Effective implementation of the ε-constraint method in multi-objective mathematical programming problems. *Applied Mathematics and Computation*, 213(2), 455-465. doi:10.1016/j.amc.2009.03.037.

Mavrotas, G., and Florios, K. (2013). An improved version of the augmented ε-constraint method (AUGMECON2) for finding the exact Pareto set in multi-objective integer programming problems. *Applied Mathematics and Computation*, 219(18), 9652-9669. doi:10.1016/j.amc.2013.03.002. (Related extension; not the same method.)

See Also

set_method_epsilon_constraint(), set_method_weighted_sum(), set_runs_grid(), set_runs_manual(), set_runs_control(), get_objectives(), plot_tradeoff(), plot_spatial_actions(), solve()

Examples

```
# EXAMPLE: Maximise woodland improvement under alternative cost limits.
# Use the same 64-unit landscape and objectives as in the other MO methods.
sim <- load_sim_multiaction()

base <- create_problem(
  pu = sim$planning_units,
  features = sim$features,
  dist_features = sim$dist_features,
  cost = "cost"
) |>
  add_actions(sim$actions, cost = sim$action_costs) |>
  add_effects(sim$effect_assumptions) |>
  # Require some improvement to avoid an empty low-cost solution.
  add_constraint_targets_relative(0.05, features = "woodland") |>
  add_objective_max_effect(
    features = "woodland", alias = "woodland"
  ) |>
  add_objective_min_cost(
    include_pu_cost = FALSE, alias = "cost"
  )

# Explore five cost ceilings. Small slack rewards help distinguish
# plans with similar woodland improvements but different costs.
augmecon <- set_method_augmecon(
  base,
  primary = "woodland",
  aliases = c("woodland", "cost"),
  runs = set_runs_grid(n = 5),
  lexicographic = TRUE,
  augmentation = 1e-3
)

if (requireNamespace("rcbc", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  solutions <- solve(
    set_solver_cbc(augmecon, time_limit = 30, verbose = FALSE)
  )

  # Actual cost may be lower than the epsilon ceiling.
  get_objectives(solutions, format = "wide")

  # Inspect the trade-off frontier generated by AUGMECON.
  plot_tradeoff(
    solutions, objectives = c("cost", "woodland"),
    label_runs = TRUE
  )
```

```
  # Compare selected management actions for three epsilon levels.
  # The three runs need not produce three different spatial plans.
  plot_spatial_actions(
    solutions, solutions = c(1, 3, 5), layout = "single", ncol = 3
  )
}

# ALTERNATIVE: use three manually specified cost ceilings.
# Start again from 'base', before configuring a multi-objective method.
manual <- set_method_augmecon(
  base,
  primary = "woodland",
  aliases = c("woodland", "cost"),
  runs = set_runs_manual(data.frame(eps_cost = c(10, 20, 40))),
  augmentation = 1e-3
)
```

`set_method_epsilon_constraint` *Configure the epsilon-constraint multi-objective method*

Description

Optimise one registered objective while imposing performance thresholds on the others. By varying these thresholds, the epsilon-constraint method produces alternative spatial plans along an observed trade-off frontier.

Usage

```
set_method_epsilon_constraint(
  x,
  primary,
  aliases = NULL,
  runs = NULL,
  eps = NULL,
  mode = NULL,
  n_points = NULL,
  include_extremes = NULL,
  lexicographic = TRUE,
  lexicographic_tol = 1e-08,
  control = NULL
)
```

Arguments

`x`	A `Problem` with at least two registered objective aliases.

`primary` Character scalar naming the objective to optimise directly.

`aliases` Optional character vector of registered objective aliases. By default, all registered objectives are used; `primary` must be included.

`runs` A run design from `set_runs_grid()` or `set_runs_manual()`. Automatic grids require exactly two objectives. Manual designs require `eps_<alias>` columns for all constrained objectives.

`eps`, `mode`, `n_points`, `include_extremes`
Deprecated arguments from the previous interface. Use `runs` instead. `include_extremes` is ignored; automatic grids always include their endpoints.

`lexicographic` Logical. Whether to refine automatic-grid anchor solutions lexicographically; the default is `TRUE`.

`lexicographic_tol`
Non-negative tolerance for lexicographic anchor refinement, expressed in the first objective's original units.

`control` Optional run-failure settings from `set_runs_control()`.

Details

First, register at least two objectives with distinct `alias` values. `primary` identifies the objective to optimise directly; every other alias included in `aliases` becomes an epsilon-constrained objective. For example, maximising woodland improvement while restricting expenditure gives

$$\max f_{\text{woodland}}(x)$$

subject to

$$f_{\text{cost}}(x) \leq \varepsilon.$$

For a constrained maximisation objective, the corresponding requirement is a lower bound, $f_k(x) \geq \varepsilon_k$. Epsilon values are objective values in their original units, not weights or relative preferences.

Automatic grids. `runs = set_runs_grid(n = 5)` generates five epsilon levels, including the two endpoints, from objective extreme-point or payoff information computed during `solve()`. Automatic grids currently support exactly **two objectives**: one primary and one constrained objective. With `lexicographic = TRUE`, the anchor solutions are refined using the other objective to break near-ties. `lexicographic_tol` controls the allowed deterioration of the first objective during that refinement; it is expressed in that objective's units. This option concerns automatic-grid anchors, not every subsequent epsilon-constrained run.

Automatic warm starts. Successful solutions are reused as initial candidates for subsequent anchor stages and epsilon runs when supported by the solver (CBC and Gurobi). Automatic grids proceed from the most restrictive secondary bound to the most permissive; manual designs keep their supplied order. A run without a usable solution retains the last successful candidate. This behaviour is internal and requires no additional argument.

Manual thresholds. Use `runs = set_runs_manual()` to supply explicit bounds. Each row defines one run, with a column named `eps_<alias>` for every constrained objective. For example, `eps_cost = 10` means *cost at most 10*, even if the optimal plan spends less. Manual designs support **two or more objectives**; they do not automatically expand combinations of epsilon levels. Use `expand.grid()` to create such a design explicitly.

Interpreting results. The method does not guarantee a different plan for each epsilon level, nor does a finite grid enumerate the full Pareto frontier. A basic epsilon-constraint formulation can also

produce weakly efficient solutions when an unconstrained secondary objective could improve without worsening the primary objective. For augmented tie-breaking, see `set_method_augmecon()`. Some requested thresholds may be infeasible; use `set_runs_control()` to specify how unsuccessful runs are handled. The returned solution diagnostics include checks of the epsilon bounds.

Only one multi-objective method can be configured on a `Problem`. To compare methods or threshold designs, start again from the problem before its method was configured. The older `eps`, `mode`, `n_points`, and `include_extremes` arguments are deprecated; use `runs` instead.

Value

An updated `Problem` with the method stored in `x$data$method`. Run-specific epsilon bounds are constructed or read during `solve()`.

References

Haimes, Y. Y., Lasdon, L. S., and Wismer, D. A. (1971). On a bicriterion formulation of the problems of integrated system identification and system optimization. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-1(3), 296-297. doi:10.1109/TSMC.1971.4308298.

Mavrotas, G. (2009). Effective implementation of the epsilon-constraint method in multi-objective mathematical programming problems. *Applied Mathematics and Computation*, 213(2), 455-465. doi:10.1016/j.amc.2009.03.037.

See Also

`set_method_weighted_sum()`, `set_method_augmecon()`, `set_runs_grid()`, `set_runs_manual()`, `set_runs_control()`, `get_objectives()`, `plot_tradeoff()`, `plot_spatial_actions()`, `solve()`

Examples

```
# EXAMPLE: Maximise woodland improvement under alternative cost limits.
# The landscape has 64 planning units and two candidate actions per unit.
sim <- load_sim_multiaction()

base <- create_problem(
  pu = sim$planning_units,
  features = sim$features,
  dist_features = sim$dist_features,
  cost = "cost"
) |>
  add_actions(sim$actions, cost = sim$action_costs) |>
  add_effects(sim$effect_assumptions) |>
  # Require some woodland improvement, so the low-cost plan is not empty.
  add_constraint_targets_relative(0.05, features = "woodland") |>
  add_objective_max_effect(
    features = "woodland", alias = "woodland"
  ) |>
  add_objective_min_cost(
    include_pu_cost = FALSE, alias = "cost"
  )

# Generate five cost limits automatically from the objective extremes.
```

```
epsilon <- set_method_epsilon_constraint(
  base,
  primary = "woodland",
  aliases = c("woodland", "cost"),
  runs = set_runs_grid(n = 5),
  lexicographic = TRUE
)

if (requireNamespace("rcbc", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  solutions <- solve(
    set_solver_cbc(epsilon, time_limit = 30, verbose = FALSE)
  )

  # Each run maximises woodland gain subject to its own cost ceiling.
  get_objectives(solutions, format = "wide")

  # The horizontal axis shows actual cost, not the epsilon ceiling.
  plot_tradeoff(
    solutions, objectives = c("cost", "woodland"),
    label_runs = TRUE
  )

  # Compare low-, intermediate- and high-cost spatial prescriptions.
  # Epsilon levels can sometimes produce the same selected actions.
  plot_spatial_actions(
    solutions, solutions = c(1, 3, 5), layout = "single", ncol = 3
  )
}

# ALTERNATIVE: specify explicit cost ceilings rather than a grid.
# Start from 'base', since each Problem can have only one method.
manual <- set_method_epsilon_constraint(
  base,
  primary = "woodland",
  aliases = c("woodland", "cost"),
  runs = set_runs_manual(data.frame(eps_cost = c(10, 20, 40)))
)
```

`set_method_weighted_sum` *Configure the weighted-sum multi-objective method*

Description

Combine two or more registered objectives using alternative non-negative weights. Each weight combination defines one optimisation run, allowing users to explore trade-offs between objectives such as implementation cost and ecological improvement.

Usage

```
set_method_weighted_sum(
  x,
  aliases,
  runs = NULL,
  weights = NULL,
  normalize_weights = NULL,
  objective_scaling = FALSE,
  control = NULL
)
```

Arguments

x	A Problem with at least two registered objective aliases.
aliases	Character vector of at least two distinct objective aliases.
runs	A run design from set_runs_grid() or set_runs_manual(). Manual designs contain columns named weight_<alias> for each selected objective.
weights	Deprecated numeric weight vector, in aliases order. Use runs = set_runs_manual(...) instead.
normalize_weights	Logical or NULL. With NULL, use TRUE for a grid and FALSE for manual weights.
objective_scaling	Logical. Request scaling of objective values before weighted aggregation. The default is FALSE.
control	Optional run-failure settings from set_runs_control().

Details

First, register at least two objectives with distinct alias values. The aliases argument selects which objectives to combine. For each run, **multiscape** constructs a weighted scalar criterion and solves the resulting mixed-integer optimisation problem.

If $f_k(x)$ is a registered objective and $w_k \geq 0$ is its weight, the scalarisation combines terms of the form $w_k f_k(x)$. Minimisation and maximisation directions are reconciled internally: larger values remain preferable for maximisation objectives, smaller values for minimisation objectives. Weights express preferences, not objective values or targets.

Selecting weights. Use runs = set_runs_grid(n = 5) to generate five combinations for two objectives, including the two single- objective extremes. Use set_runs_manual() to provide exact weights in columns named weight_<alias>; each row is a separate run.

Normalisation versus objective scaling. normalize_weights rescales the weights within each run to sum to one. By default, grid weights are normalised, whereas manual weights are retained as supplied. This does not make objectives with different units numerically comparable. objective_scaling = TRUE instead requests scaling of the objective values before aggregation, so the preferences are less sensitive to their numerical ranges. The scaling is performed when solving; reported objective values retain their original units.

Interpreting solutions. Different weights can produce the same spatial plan, particularly in discrete optimisation problems. Weighted sums generally recover supported efficient solutions; they

need not find all non-dominated solutions of a non-convex frontier. For a more systematic exploration, consider `set_method_epsilon_constraint()` or `set_method_augmecon()`. Zero-weight runs may also have ties that are not resolved with respect to the omitted objective.

The `control` argument specifies how to handle unsuccessful runs. A problem can have only one configured multi-objective method; to compare methods or weight designs, begin from the same problem before calling a `set_method_*()` function. The deprecated `weights` argument is still accepted for compatibility, but `runs` is recommended.

Value

An updated `Problem` with the method stored in `x$data$method`. The optimisation runs are executed by `solve()`.

References

Ehrgott, M. (2005). *Multicriteria Optimization*, 2nd ed. Springer. doi:10.1007/3540276599.

Miettinen, K. (1998). *Nonlinear Multiobjective Optimization*. Kluwer Academic Publishers. doi:10.1007/9781461555636.

See Also

set_runs_grid, set_runs_manual, set_runs_control, set_method_epsilon_constraint, set_method_augmecon, plot_tradeoff, plot_spatial_actions, solve

Examples

```
# EXAMPLE: Trade off management costs against woodland improvement.
# The simulated landscape has 64 planning units and two possible actions.
sim <- load_sim_multiaction()

base <- create_problem(
  pu = sim$planning_units,
  features = sim$features,
  dist_features = sim$dist_features,
  cost = "cost"
) |>
  add_actions(sim$actions, cost = sim$action_costs) |>
  add_effects(sim$effect_assumptions) |>
  # Require some improvement, so the cheapest plan is not empty.
  add_constraint_targets_relative(0.05, features = "woodland") |>
  add_objective_min_cost(
    include_pu_cost = FALSE, alias = "cost"
  ) |>
  add_objective_max_effect(
    features = "woodland", alias = "woodland"
  )

# Five different preferences, from cost-only to woodland-only.
weighted <- set_method_weighted_sum(
  base,
  aliases = c("cost", "woodland"),
```

```
  runs = set_runs_grid(n = 5),
  objective_scaling = TRUE
)

if (requireNamespace("rcbc", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  solutions <- solve(
    set_solver_cbc(weighted, time_limit = 30, verbose = FALSE)
  )

  # Inspect the original objective values across runs.
  get_objectives(solutions, format = "wide")

  # Explore the observed cost-improvement trade-off.
  plot_tradeoff(
    solutions, objectives = c("cost", "woodland"),
    label_runs = TRUE
  )

  # Compare three spatial prescriptions in the same figure.
  # These are preferences, not necessarily three distinct plans.
  plot_spatial_actions(
    solutions, solutions = c(1, 3, 5), layout = "single", ncol = 3
  )
}

# ALTERNATIVE: supply three explicit weight combinations.
# Reuse 'base', because a Problem can have only one method.
manual <- set_method_weighted_sum(
  base,
  aliases = c("cost", "woodland"),
  runs = set_runs_manual(data.frame(
    weight_cost = c(1, 0.5, 0),
    weight_woodland = c(0, 0.5, 1)
  )),
  objective_scaling = TRUE
)
```

`set_runs_control` *Control multi-objective run behavior*

Description

Create a control object that determines how multi-objective workflows respond to infeasible runs, missing solution vectors, and unexpected errors.

The resulting object is supplied to the `control` argument of `set_method_epsilon_constraint` or `set_method_augmecon`.

Usage

```
set_runs_control(
  stop_on_infeasible = FALSE,
  stop_on_no_solution = FALSE,
  stop_on_error = TRUE
)
```

Arguments

stop_on_infeasible	Logical. If TRUE, stop the complete multi-objective workflow when a run is reported as infeasible. If FALSE, retain the attempted run in the run table and continue with the remaining runs. Defaults to FALSE.
stop_on_no_solution	Logical. If TRUE, stop when a run does not return a usable solution vector. If FALSE, retain the attempted run without a stored solution_id and continue. Defaults to FALSE.
stop_on_error	Logical. If TRUE, stop on unexpected errors raised during model preparation, solving, or result processing. If FALSE, attempt to record the failed run and continue. Defaults to TRUE.

Details

Multi-objective methods commonly solve a sequence of related optimization models. Some parameter combinations, particularly restrictive epsilon levels, may be infeasible or may fail to produce a usable solution.

set_runs_control() determines whether such events stop the entire multi-objective workflow or are recorded in the resulting solutionset-class object.

A SolutionSet distinguishes between:

- a run_id, which identifies every attempted optimization run;
- a solution_id, which is assigned only when a run produces a stored solution.

When a run is retained after an infeasibility or missing-solution event, its run-level metadata remains available through get_runs, but its solution_id and objective values will generally be missing.

Infeasible runs

If stop_on_infeasible = FALSE, an infeasible run is recorded and the workflow continues with the remaining run design. This is generally useful when exploring automatically generated epsilon grids because restrictive combinations of epsilon levels may be infeasible.

If stop_on_infeasible = TRUE, the workflow stops when the first infeasible run is encountered.

Runs without a solution vector

A solver may terminate without returning a usable decision vector even when the outcome is not classified explicitly as infeasible.

If stop_on_no_solution = FALSE, the attempted run is recorded without a stored solution and the remaining runs are attempted. If stop_on_no_solution = TRUE, the workflow stops immediately.

Unexpected errors

If `stop_on_error = TRUE`, unexpected errors raised while preparing, solving, or processing a run are propagated and stop the workflow. This is the recommended default because such errors may indicate an invalid model, unsupported solver behaviour, or an internal implementation problem.

If `stop_on_error = FALSE`, the workflow attempts to record the failed run and continue. This option should be used cautiously because it may conceal modelling or implementation errors.

The default settings favour completing the requested run design while still stopping on unexpected errors:

```
stop_on_infeasible = FALSE
stop_on_no_solution = FALSE
stop_on_error = TRUE
```

Value

An object of class `RunsControl` and `MOControl` containing the validated execution-control settings. The object is intended to be supplied to the `control` argument of a supported multi-objective method.

See Also

`set_method_epsilon_constraint`, `set_method_augmecon`, `set_runs_grid`, `set_runs_manual`, `get_runs`, `solutionset-class`

Examples

```
# Default behaviour: continue after infeasible runs or runs without a
# solution, but stop on unexpected errors
control <- set_runs_control()
control

# Stop as soon as an infeasible run or missing solution is encountered
strict_control <- set_runs_control(
  stop_on_infeasible = TRUE,
  stop_on_no_solution = TRUE
)

strict_control
```

---

set_runs_grid *Define an automatic multi-objective run grid*

Description

Create an automatic run-design specification for generating multiple optimization runs in a multi-objective workflow.

`set_runs_grid()` provides a common interface for controlling the resolution of weighted-sum, epsilon-constraint, and AUGMECON run designs. The returned object does not contain the final run table. Instead, it stores the requested grid resolution, which is resolved later by the corresponding `set_method_*()` function using the registered objectives and their optimization senses.

Usage

```
set_runs_grid(n)
```

Arguments

`n`	Integer. Resolution of the automatically generated run design. Must be at least `2`. The final number of optimization runs may differ from `n`, depending on the selected method and the number of objectives.

Details

Multi-objective methods generally require several optimization runs to explore different regions of objective space. `set_runs_grid()` asks **multiscape** to generate those runs automatically from a grid.

The interpretation of the grid depends on the selected method:

- In `set_method_weighted_sum`, the grid defines combinations of objective weights. For two objectives, `n` gives the number of weight combinations between the two pure-objective extremes. For three or more objectives, `n` controls the resolution of the generated simplex grid. The generated weights are normalized according to the method settings and represent alternative preferences among the registered objectives.
- In `set_method_epsilon_constraint`, the grid defines epsilon levels for the constrained objective. The primary objective is optimized directly, while the remaining objective is progressively restricted across the generated runs.
- In `set_method_augmecon`, the grid defines epsilon levels for the secondary objectives, which are then used in the augmented epsilon-constraint formulation. When several secondary objectives are used, the final run design is obtained by combining the generated epsilon levels across objectives.

The argument `n` controls the resolution of the automatically generated design. It should not always be interpreted as the final number of runs. For example, when several secondary objectives are present, epsilon levels may be combined across objectives and generate more runs than the value supplied to `n`. Likewise, the number of valid weighted combinations depends on the number of objectives and on how the weight grid is constructed.

Boundary levels are always included. For weighted-sum methods, this means that pure-objective weight vectors are included, where all weight is assigned to one objective. For epsilon-constraint and AUGMECON methods, this means that the lower and upper bounds of the automatically derived epsilon ranges are included.

Including boundary levels helps recover the best observed value of each objective and provides reference points for subsequent frontier analyses.

The resolved design is stored in the resulting solutionset-class object and can be inspected after solving through get_runs.

Use set_runs_manual instead when exact weights or epsilon levels must be supplied explicitly.

Value

An object of class RunGrid and RunDesign. The object stores the requested grid resolution and is intended to be supplied to the runs argument of a multi-objective method function.

See Also

set_runs_manual, set_method_weighted_sum, set_method_epsilon_constraint, set_method_augmecon, get_runs

Examples

```
# Create an automatic run-grid specification
grid <- set_runs_grid(n = 5)

grid

# Use the automatic grid with a complete simulated planning problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
  add_objective_max_effect(alias = "benefit") |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = grid,
    normalize_weights = TRUE
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(
    problem,
```

```
  verbose = FALSE
)

solutions <- solve(problem)

# Inspect the resolved run design and objective values
get_runs(solutions)
get_objectives(
  solutions,
  format = "wide"
)
}
```

`set_runs_manual` *Define a manual multi-objective run design*

Description

Create an explicit run-design specification in which each row defines one multi-objective optimization run.

`set_runs_manual()` is used when exact objective weights or epsilon levels should be supplied by the user instead of being generated automatically with `set_runs_grid`.

Usage

```
set_runs_manual(x)
```

Arguments

`x`	A non-empty `data.frame` with one row per optimization run. Run-design columns must be named using the `weight_<alias>` or `eps_<alias>` convention.

Details

The input must be a non-empty `data.frame` with one row per requested optimization run.

Weighted-sum columns must follow the convention:

`weight_<alias>`

Epsilon-constraint and AUGMECON columns must follow the convention:

`eps_<alias>`

A manual design must contain at least one column beginning with `weight_` or `eps_`. Mixing both column families in the same design is not allowed.

All run-design columns must be numeric, finite, and free of missing values. Weight columns must contain non-negative values, and each weighted-sum row must assign a strictly positive total weight.

This function performs method-independent structural validation. Method-specific validation is performed later by the corresponding `set_method_*()` function. This includes checking:

- that all required objective aliases are represented;
- that no unknown objective columns are supplied;
- that epsilon columns correspond only to secondary objectives;
- and that the supplied design is compatible with the selected method.

Therefore, an object may be structurally valid for `set_runs_manual()` but rejected later by `set_method_weighted_sum`, `set_method_epsilon_constraint`, or `set_method_augmecon`.

Value

An object of class `RunManual` and `RunDesign` containing the validated run table.

See Also

`set_runs_grid`, `set_method_weighted_sum`, `set_method_epsilon_constraint`, `set_method_augmecon`

Examples

```
weighted_runs <- set_runs_manual(
  data.frame(
    weight_cost = c(1.0, 0.5, 0.0),
    weight_benefit = c(0.0, 0.5, 1.0)
  )
)

epsilon_runs <- set_runs_manual(
  data.frame(
    eps_benefit = c(2, 4, 6, 8)
  )
)

weighted_runs
epsilon_runs
```

`set_solver` *Configure solver settings*

Description

Store solver configuration inside a `Problem` object so that `solve` can later run using the stored backend and runtime options.

This function does not build or solve the optimization model. It only updates the solver configuration stored in `x$data$solve_args`.

Usage

```
set_solver(
  x,
  solver = NULL,
  gap_limit = NULL,
  time_limit = NULL,
  solution_limit = NULL,
  cores = NULL,
  verbose = NULL,
  log_file = NULL,
  write_log = NULL,
  solver_params = list(),
  ...
)
```

Arguments

`x`	A `Problem` object.
`solver`	Character string indicating the solver backend to use. Must be one of `"auto"`, `"gurobi"`, `"cplex"`, `"cbc"`, or `"symphony"`.
`gap_limit`	Optional numeric value in $[0, 1]$ giving the relative optimality gap for mixed-integer optimization. If `NULL`, the previously stored value is kept unchanged.
`time_limit`	Optional non-negative numeric value giving the maximum solving time in seconds. If `NULL`, the previously stored value is kept unchanged.
`solution_limit`	Optional logical flag requesting early termination after a feasible solution is found. Supported by Gurobi, CBC, and SYMPHONY, but not by CPLEX through Rcplex. If `NULL`, the previously stored value is kept unchanged.
`cores`	Optional positive integer giving the maximum number of solver threads. Currently supported by Gurobi. If `NULL`, the previously stored value is kept unchanged.
`verbose`	Optional logical flag indicating whether the solver should print log output. If `NULL`, the previously stored value is kept unchanged.

log_file	Optional character string giving the complete path or file name of the solver log. Currently supported by Gurobi. If NULL, the previously stored value is kept unchanged.
write_log	Optional logical flag indicating whether solver output should be written to a file. Currently supported by Gurobi. If NULL, the previously stored value is kept unchanged.
solver_params	Named list of solver-specific parameters. These are merged with previously stored parameters. Rcplex parameters are validated against its supported control names; Rsymphony does not currently receive arbitrary solver-specific parameters.
...	Additional named solver-specific parameters. These are merged into solver_params. For example, MIPFocus = 1 for Gurobi.

Details

Purpose

The multiscape workflow separates problem specification from solver configuration. Problem data, actions, effects, targets, objectives, and methods are stored in the Problem object, and solver settings are stored separately in x$data$solve_args.

This function allows solver options to be configured once and reused later through solve(x) without repeating the same arguments each time.

Stored fields

The solver configuration is stored in x$data$solve_args. Typical entries include:

- solver,
- gap_limit,
- time_limit,
- solution_limit,
- cores,
- verbose,
- write_log,
- log_file,
- solver_params.

Single configuration

Solver settings can be configured only once per problem, including through backend wrappers. A second call raises an error, even if the arguments are identical. Supply all settings together. Arguments left as NULL use solver defaults. Reading defaults does not count as explicit configuration. To compare configurations, derive each alternative from the same problem before the first solver call.

Gap limit

The argument gap_limit is interpreted as a relative optimality gap for mixed-integer optimization. It must lie in $[0, 1]$.

If the solver stops with incumbent value z^{inc} and best bound z^{bd}, then the exact stopping rule depends on the solver backend, but conceptually `gap_limit` controls the maximum accepted relative difference between the incumbent and the bound.

Time limit

The argument `time_limit` is interpreted as a maximum wall-clock time in seconds allowed for the solver.

Solution limit

The argument `solution_limit` is stored as a logical flag. Its exact meaning depends on the backend-specific solving layer, but conceptually it requests early termination after finding a feasible solution according to the behaviour supported by the chosen solver.

Cores

The argument `cores` specifies the maximum number of solver threads. It is supported by Gurobi. Rcplex, rcbc, and Rsymphony do not reliably expose thread control through the interfaces used here; multiscape therefore warns and ignores `cores` for CPLEX, CBC, and SYMPHONY. If the requested number exceeds the number of detected logical processors, it is capped to the detected maximum with a warning.

Verbose output and log files

The arguments `verbose`, `write_log`, and `log_file` control how solver logging is handled. These options are stored and later interpreted by the solving layer for the selected backend. Solver log files are currently available only with Gurobi. A parameter that is not available through the selected R solver interface is reported with a warning and is not stored as if it had been applied.

Backend capabilities

Common parameters are translated only when the selected backend supports them:

- Gurobi supports `cores`, `solution_limit`, and `write_log`.
- CBC supports `solution_limit`. Thread control and CBC log files are not exposed reliably through the current rcbc integration.
- CPLEX through Rcplex does not expose `cores`, `solution_limit`, or solver log files.
- SYMPHONY through Rsymphony supports `solution_limit`, but does not expose `cores` or solver log files.

Solver-specific parameters

Additional backend-specific parameters can be passed in two ways:

- through the named list `solver_params`,
- through additional named arguments in `...`.

These two sources are merged within the single configuration call.

This is particularly useful for backend-specific controls such as node selection, emphasis parameters, tolerances, or heuristics.

Supported backends

The `solver` argument selects the backend to be used later by `solve`. Supported values are:

- `"auto"`: let the solving layer choose an available backend,

- "gurobi",
- "cplex",
- "cbc",
- "symphony".

This function only stores the requested backend. Availability of the backend is checked later when solving.

Value

An updated Problem object with modified solver settings stored in x$data$solve_args.

See Also

solve, set_solver_gurobi, set_solver_cplex, set_solver_cbc, set_solver_symphony

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

x <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
)

x1 <- set_solver(
  x,
  solver = "cbc",
  gap_limit = 0.01,
  time_limit = 300,
  cores = 2,
  verbose = TRUE
)

x1$data$solve_args

# Define an alternative from the common unconfigured problem
x2 <- set_solver(
  x,
  gap_limit = 0.05,
  solver_params = list(randomSeed = 123)
)

x2$data$solve_args
```

set_solver_cbc *Configure CBC solver settings*

Description

Convenience wrapper around set_solver that stores solver = "cbc" in the problem object.

This function does not solve the model. It only updates the stored solver configuration. A solver can be configured only once per problem. Calling this wrapper after any solver setter raises an error. Derive alternatives from the same problem before its first solver configuration.

Usage

```
set_solver_cbc(
  x,
  ...,
  solver_params = list(),
  gap_limit = NULL,
  time_limit = NULL,
  solution_limit = NULL,
  cores = NULL,
  verbose = NULL,
  log_file = NULL,
  write_log = NULL
)
```

Arguments

x	A Problem object.
...	Additional named solver-specific parameters. These are merged into solver_params. For example, MIPFocus = 1 for Gurobi.
solver_params	Named list of solver-specific parameters. These are merged with previously stored parameters. Rcplex parameters are validated against its supported control names; Rsymphony does not currently receive arbitrary solver-specific parameters.
gap_limit	Optional numeric value in $[0, 1]$ giving the relative optimality gap for mixed-integer optimization. If NULL, the previously stored value is kept unchanged.
time_limit	Optional non-negative numeric value giving the maximum solving time in seconds. If NULL, the previously stored value is kept unchanged.
solution_limit	Optional logical flag requesting early termination after a feasible solution is found. Supported by Gurobi, CBC, and SYMPHONY, but not by CPLEX through Rcplex. If NULL, the previously stored value is kept unchanged.
cores	Optional positive integer giving the maximum number of solver threads. Currently supported by Gurobi. If NULL, the previously stored value is kept unchanged.

verbose	Optional logical flag indicating whether the solver should print log output. If NULL, the previously stored value is kept unchanged.
log_file	Optional character string giving the complete path or file name of the solver log. Currently supported by Gurobi. If NULL, the previously stored value is kept unchanged.
write_log	Optional logical flag indicating whether solver output should be written to a file. Currently supported by Gurobi. If NULL, the previously stored value is kept unchanged.

Value

An updated Problem object with CBC solver settings stored in x$data$solve_args.

See Also

set_solver, solve

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

x <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
)

x <- set_solver_cbc(
  x,
  gap_limit = 0.01,
  time_limit = 300,
  cores = 2,
  solution_limit = FALSE
)

x$data$solve_args
```

---

set_solver_cplex *Configure CPLEX solver settings*

Description

Convenience wrapper around set_solver that sets solver = "cplex". A solver can be configured only once per problem. Calling this wrapper after any solver setter raises an error. Derive alternatives from the same problem before its first solver configuration.

Usage

```
set_solver_cplex(
  x,
  ...,
  solver_params = list(),
  gap_limit = NULL,
  time_limit = NULL,
  solution_limit = NULL,
  cores = NULL,
  verbose = NULL,
  log_file = NULL,
  write_log = NULL
)
```

Arguments

x	A Problem object.
...	Additional named solver-specific parameters. These are merged into solver_params. For example, MIPFocus = 1 for Gurobi.
solver_params	Named list of solver-specific parameters. These are merged with previously stored parameters. Rcplex parameters are validated against its supported control names; Rsymphony does not currently receive arbitrary solver-specific parameters.
gap_limit	Optional numeric value in $[0, 1]$ giving the relative optimality gap for mixed-integer optimization. If NULL, the previously stored value is kept unchanged.
time_limit	Optional non-negative numeric value giving the maximum solving time in seconds. If NULL, the previously stored value is kept unchanged.
solution_limit	Optional logical flag requesting early termination after a feasible solution is found. Supported by Gurobi, CBC, and SYMPHONY, but not by CPLEX through Rcplex. If NULL, the previously stored value is kept unchanged.
cores	Optional positive integer giving the maximum number of solver threads. Currently supported by Gurobi. If NULL, the previously stored value is kept unchanged.
verbose	Optional logical flag indicating whether the solver should print log output. If NULL, the previously stored value is kept unchanged.
log_file	Optional character string giving the complete path or file name of the solver log. Currently supported by Gurobi. If NULL, the previously stored value is kept unchanged.
write_log	Optional logical flag indicating whether solver output should be written to a file. Currently supported by Gurobi. If NULL, the previously stored value is kept unchanged.

Value

An updated Problem object with CPLEX solver settings.

See Also

set_solver, solve

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

x <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
)

x <- set_solver_cplex(
  x,
  gap_limit = 0.001,
  time_limit = 1200
)

x$data$solve_args
```

---

set_solver_gurobi *Configure Gurobi solver settings*

Description

Convenience wrapper around set_solver that stores solver = "gurobi" in the problem object.

This function does not solve the model. It only updates the stored solver configuration. A solver can be configured only once per problem. Calling this wrapper after any solver setter raises an error. Derive alternatives from the same problem before its first solver configuration.

Usage

```
set_solver_gurobi(
  x,
  ...,
  solver_params = list(),
  gap_limit = NULL,
  time_limit = NULL,
  solution_limit = NULL,
  cores = NULL,
  verbose = NULL,
  log_file = NULL,
  write_log = NULL
)
```

Arguments

`x`	A `Problem` object.
`...`	Additional named solver-specific parameters. These are merged into `solver_params`. For example, `MIPFocus = 1` for Gurobi.
`solver_params`	Named list of solver-specific parameters. These are merged with previously stored parameters. Rcplex parameters are validated against its supported control names; Rsymphony does not currently receive arbitrary solver-specific parameters.
`gap_limit`	Optional numeric value in $[0, 1]$ giving the relative optimality gap for mixed-integer optimization. If `NULL`, the previously stored value is kept unchanged.
`time_limit`	Optional non-negative numeric value giving the maximum solving time in seconds. If `NULL`, the previously stored value is kept unchanged.
`solution_limit`	Optional logical flag requesting early termination after a feasible solution is found. Supported by Gurobi, CBC, and SYMPHONY, but not by CPLEX through Rcplex. If `NULL`, the previously stored value is kept unchanged.
`cores`	Optional positive integer giving the maximum number of solver threads. Currently supported by Gurobi. If `NULL`, the previously stored value is kept unchanged.
`verbose`	Optional logical flag indicating whether the solver should print log output. If `NULL`, the previously stored value is kept unchanged.
`log_file`	Optional character string giving the complete path or file name of the solver log. Currently supported by Gurobi. If `NULL`, the previously stored value is kept unchanged.
`write_log`	Optional logical flag indicating whether solver output should be written to a file. Currently supported by Gurobi. If `NULL`, the previously stored value is kept unchanged.

Value

An updated `Problem` object with Gurobi solver settings stored in `x$data$solve_args`.

See Also

`set_solver`, `solve`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

x <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
)
```

```
x <- set_solver_gurobi(
  x,
  gap_limit = 0.01,
  time_limit = 600,
  cores = 2,
  MIPFocus = 1
)

x$data$solve_args
```

`set_solver_symphony` *Configure SYMPHONY solver settings*

Description

Convenience wrapper around `set_solver` that stores `solver = "symphony"` in the problem object.

This function does not solve the model. It only updates the stored solver configuration. A solver can be configured only once per problem. Calling this wrapper after any solver setter raises an error. Derive alternatives from the same problem before its first solver configuration.

Usage

```
set_solver_symphony(
  x,
  ...,
  solver_params = list(),
  gap_limit = NULL,
  time_limit = NULL,
  solution_limit = NULL,
  cores = NULL,
  verbose = NULL,
  log_file = NULL,
  write_log = NULL
)
```

Arguments

`x`	A `Problem` object.
`...`	Additional named solver-specific parameters. These are merged into `solver_params`. For example, `MIPFocus = 1` for Gurobi.
`solver_params`	Named list of solver-specific parameters. These are merged with previously stored parameters. Rcplex parameters are validated against its supported control names; Rsymphony does not currently receive arbitrary solver-specific parameters.
`gap_limit`	Optional numeric value in $[0, 1]$ giving the relative optimality gap for mixed-integer optimization. If `NULL`, the previously stored value is kept unchanged.

time_limit	Optional non-negative numeric value giving the maximum solving time in seconds. If NULL, the previously stored value is kept unchanged.
solution_limit	Optional logical flag requesting early termination after a feasible solution is found. Supported by Gurobi, CBC, and SYMPHONY, but not by CPLEX through Rcplex. If NULL, the previously stored value is kept unchanged.
cores	Optional positive integer giving the maximum number of solver threads. Currently supported by Gurobi. If NULL, the previously stored value is kept unchanged.
verbose	Optional logical flag indicating whether the solver should print log output. If NULL, the previously stored value is kept unchanged.
log_file	Optional character string giving the complete path or file name of the solver log. Currently supported by Gurobi. If NULL, the previously stored value is kept unchanged.
write_log	Optional logical flag indicating whether solver output should be written to a file. Currently supported by Gurobi. If NULL, the previously stored value is kept unchanged.

Value

An updated Problem object with SYMPHONY solver settings stored in x$data$solve_args.

See Also

set_solver, solve

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

x <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
)

x <- set_solver_symphony(
  x,
  gap_limit = 0.05,
  time_limit = 300
)

x$data$solve_args
```

sim_dist_features *Simulated feature distribution*

Description

Historical Meseta-related distribution with 348,021 rows, paired with `sim_pu` and `sim_features`. It is not the zero credited comparison reference used in the introductory tutorial. Prefer `load_meseta()`.

Usage

```
data(sim_dist_features)
```

Format

A data frame linking planning units and features with an `amount` column.

sim_features *Simulated features*

Description

Historical Meseta-related catalogue of 155 features, retained for compatibility. These do not belong to the compact simulated example. Prefer `load_meseta()` for the complete tutorial inputs.

Usage

```
data(sim_features)
```

Format

A data frame with feature identifiers and names.

sim_multiaction *Simulated spatial multi-action planning inputs*

Description

Prefer `load_sim_multiaction()` as the entry point to this dataset. Direct `data()` access remains available for compatibility.

Usage

```
data(sim_multiaction)
```

Format

A named list with seven components:

`planning_units` An `sf` object with 64 square planning units.

`features` A data frame with two feature identifiers and names.

`dist_features` A data frame of feature amounts by planning unit.

`actions` A data frame describing protection and restoration.

`action_costs` A data frame of spatially varying action costs.

`effect_assumptions` Relative changes by action and feature, ready to pass to `add_effects`.

`effects` Explicit signed changes in the historical `delta` column, retained for compatibility. New examples use `effect_assumptions`.

Details

A compact example dataset containing all inputs needed to build a multi-objective spatial planning problem with protection and restoration as mutually exclusive candidate actions.

---

sim_pu *Simulated planning units*

Description

Historical Meseta-related planning units: 11,109 spatial cells with unit cost 1. These are distinct from `sim_pu_sf` and are not the complete Meseta tutorial inputs. Prefer `load_meseta()` for that tutorial.

Usage

```
data(sim_pu)
```

Format

An object of class `sf`.

sim_pu_sf *Simulated planning units*

Description

Historical ecosystem-services planning units: 30,496 spatial cells. These are distinct from `sim_pu`. Prefer `load_ecosystem_services()` to load the matching planning units and feature raster.

Usage

```
data(sim_pu_sf)
```

Format

An object of class `sf`.

solution_append *Append solutions from another solution set*

Description

Append the runs and stored solutions from one `solutionset-class` object to another.

This function combines two `SolutionSet` objects generated from the same planning problem. It is intended for combining results obtained with different **multiscape** solving workflows, such as weighted-sum, epsilon-constraint, or AUGMECON methods, while keeping a single coherent result object for downstream extraction, plotting, and analysis.

Usage

```
solution_append(x, y)
```

Arguments

`x`	A `solutionset-class` object returned by `solve`.
`y`	A second `solutionset-class` object returned by `solve` and generated from the same planning problem as `x`.

Details

`solution_append()` is a solution-set management function. It does not modify the original input objects. Instead, it returns a new `SolutionSet` containing the runs and solutions from both inputs.

Both input objects must be generated from the same planning problem. This is checked conservatively before appending. In particular, the two objects must have compatible:

- planning units;

- features and feature distributions;
- actions, feasible action pairs, and effects;
- profit data, when present;
- targets;
- locks and constraints;
- spatial relations;
- objective specifications.

Differences in method settings, run design, solver settings, solver status, runtime, gaps, and other solve diagnostics are allowed.

The appended runs and solutions are assigned new `run_id` and `solution_id` values to keep identifiers unique in the combined object. Identifiers are not required to match between the two inputs.

This function is not intended to combine results from different planning problems, scenarios, target sets, or objective definitions. Such workflows should be handled by a future comparison/binding function rather than by `solution_append()`.

Value

A new `solutionset-class` object containing the runs and stored solutions from both input objects.

See Also

`solution_filter`, `solutionset-class`, `get_runs`, `get_objectives`, `plot_tradeoff`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

make_problem <- function() {
  create_problem(
    pu = example_data$planning_units,
    features = example_data$features,
    dist_features = example_data$dist_features,
    cost = "cost"
  ) |>
    add_actions(
      example_data$actions,
      cost = example_data$action_costs
    ) |>
    add_effects(
      example_data$effect_assumptions
    ) |>
    add_constraint_targets_relative(0.05) |>
    add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
    add_objective_max_effect(alias = "benefit")
}

weighted_problem <- make_problem() |>
```

```
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(
      n = 4
    ),
    normalize_weights = TRUE
  )

epsilon_problem <- make_problem() |>
  set_method_epsilon_constraint(
    primary = "cost",
    runs = set_runs_grid(
      n = 4
    )
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
  weighted_problem <- set_solver_cbc(
    weighted_problem,
    verbose = FALSE
  )

  epsilon_problem <- set_solver_cbc(
    epsilon_problem,
    verbose = FALSE
  )

  weighted_solutions <- solve(weighted_problem)
  epsilon_solutions <- solve(epsilon_problem)

  combined_solutions <- solution_append(
    weighted_solutions,
    epsilon_solutions
  )

  # Inspect the combined run history
  get_runs(combined_solutions)

  # Inspect objective values from both methods
  get_objectives(
    combined_solutions,
    format = "wide"
  )

  # The original objects remain unchanged
  get_runs(weighted_solutions)
  get_runs(epsilon_solutions)
}
```

`solution_filter` *Filter solutions in a solution set*

Description

Return a reduced solutionset-class object containing only the runs or solutions that match the requested filters.

This function is intended to curate SolutionSet objects before downstream analysis, plotting, frontier analysis, or post-hoc evaluation. It filters all relevant components of the object consistently, including the run table, design table, stored run-level solutions, and available summary tables.

Usage

```
solution_filter(
  x,
  run_id = NULL,
  solution_id = NULL,
  status = NULL,
  feasible_only = FALSE,
  nondominated = FALSE,
  objectives = NULL
)
```

Arguments

x	A solutionset-class object returned by solve.
run_id	Optional integer vector of run ids to keep.
solution_id	Optional numeric vector of positive integer solution ids to keep. Runs without a stored solution are never matched by this filter.
status	Optional character vector of run statuses to keep. Matching is case-insensitive.
feasible_only	Logical. If TRUE, keep only runs whose status is interpreted as having produced a usable solution. The current accepted statuses are "optimal", "feasible", "suboptimal", "time_limit", "time_limit_feasible", "gap_limit", "gap_limit_feasible", and "solution_limit".
nondominated	Logical. If TRUE, keep only non-dominated solutions among the runs retained by the other filters. This uses **moocore** internally.
objectives	Optional character vector of objective names to use when nondominated = TRUE. If NULL, all available objective-value columns are used.

Details

A SolutionSet distinguishes between runs and stored solutions.

- run_id identifies a run or attempted solve. Runs may be feasible, optimal, infeasible, failed, or otherwise incomplete.
- solution_id identifies a stored solution. Runs that did not produce a solution have solution_id = NA.

Therefore, filtering by run_id and filtering by solution_id are not always equivalent. For example, an infeasible run may have a run_id but no solution_id.

The function filters:

- `x$solution$runs`, using the selected `run_ids`;
- `x$solution$design`, when it contains a `run_id` column;
- `x$solution$solutions`, using the selected `solution_ids`;
- all tables in `x$summary` that contain a `run_id` column.

The function does not renumber `run_id` or `solution_id`. This preserves traceability to the original run design.

If more than one filter is supplied, filters are combined using logical *and*. For example, setting both `status = "optimal"` and `solution_id = c(1, 3)` keeps only optimal runs whose `solution_id` is either 1 or 3.

If `nondominated = TRUE`, the function further keeps only non-dominated solutions among the runs retained by the previous filters. Dominance is evaluated in objective space using the objective values stored in the run table. Objective senses are obtained from the objective specifications stored in the original problem.

Non-dominated filtering requires the **moocore** package.

Value

A filtered `solutionset-class` object.

See Also

`solutionset-class`, `solve`, `get_runs`, `get_objectives`, `get_planning_units`, `get_actions`

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
  add_objective_max_effect(alias = "benefit") |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 3),
    normalize_weights = TRUE
  )
```

```
if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(
    problem,
    verbose = FALSE
  )

  solutions <- solve(problem)
  runs <- get_runs(solutions)

  # Keep only runs with a usable solver status
  feasible_solutions <- solution_filter(
    solutions,
    feasible_only = TRUE
  )

  # Keep selected runs
  selected_runs <- solution_filter(
    solutions,
    run_id = runs$run_id[1:2]
  )

  # Keep one stored solution
  solution_ids <- runs$solution_id[
    !is.na(runs$solution_id)
  ]

  if (length(solution_ids) > 0L) {
    selected_solution <- solution_filter(
      solutions,
      solution_id = solution_ids[1]
    )
  }

  # Keep only optimal runs
  if ("optimal" %in% tolower(runs$status)) {
    optimal_solutions <- solution_filter(
      solutions,
      status = "optimal"
    )
  }

  # Keep only non-dominated solutions
  if (requireNamespace("moocore", quietly = TRUE)) {
    nondominated_solutions <- solution_filter(
      solutions,
      feasible_only = TRUE,
      nondominated = TRUE
    )

    # Evaluate dominance using selected objectives
    nondominated_subset <- solution_filter(
      solutions,
```

```
      feasible_only = TRUE,
      nondominated = TRUE,
      objectives = c("cost", "benefit")
    )
  }
}
```

`solution_unique` *Keep unique solutions in a solution set*

Description

Return a reduced `solutionset-class` object containing one representative from each group of equivalent solutions.

Solutions can be considered equivalent according to either their decision vectors or their objective values.

Usage

```
solution_unique(
  x,
  by = c("decisions", "objectives"),
  keep = c("first", "last"),
  objectives = NULL,
  tolerance = sqrt(.Machine$double.eps)
)
```

Arguments

`x`	A `solutionset-class` object returned by `solve`.
`by`	Character. Definition of uniqueness. One of `"decisions"` or `"objectives"`.
`keep`	Character. Which representative to retain from each group of equivalent solutions. If `"first"`, retain the first solution in the current run order. If `"last"`, retain the last.
`objectives`	Optional character vector of objective names to compare when `by = "objectives"`. If `NULL`, all available objectives are used. This argument is ignored when `by = "decisions"`.
`tolerance`	Non-negative numeric tolerance used when comparing objective values. It is only used when `by = "objectives"`.

Details

`solution_unique()` is a solution-set management function. It removes repeated solutions while consistently filtering the run table, design table, stored run-level solutions, and summary tables.

Two definitions of uniqueness are supported:

- by = "decisions" compares the complete stored decision vector of each solution. Two solutions are considered equivalent when their decision vectors are identical. This identifies repeated planning-unit or planning-unit/action configurations, even when they were generated by different runs or multi-objective parameter combinations.
- by = "objectives" compares the selected objective values. Two solutions are considered equivalent when all compared objective values are equal within the specified numerical tolerance. Such solutions may still have different decision vectors.

Consequently, uniqueness in objective space and uniqueness in decision space are not equivalent. Two spatially different solutions may produce the same objective values, while repeated runs may generate exactly the same decision vector.

Only runs with a stored solution_id can be assessed. Runs without a stored solution, such as infeasible runs, are preserved unchanged. This retains the full run history while removing only duplicated stored solutions.

The function does not renumber run_id or solution_id. The representative retained from each duplicate group keeps its original identifiers, preserving traceability to the original run design.

For by = "objectives", numerical equality is assessed using a relative comparison:

$$|a - b| \leq \epsilon \max(1, |a|, |b|),$$

where ϵ is specified by tolerance. This avoids treating insignificant floating-point differences as distinct objective points.

Value

A new solutionset-class object containing one representative from each group of equivalent stored solutions, together with any runs that did not produce a stored solution.

See Also

solution_filter, solution_append, get_objectives,

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

problem <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
```

```
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost", include_pu_cost = FALSE) |>
  add_objective_max_effect(alias = "benefit") |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = set_runs_grid(n = 3),
    normalize_weights = TRUE
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
  problem <- set_solver_cbc(
    problem,
    verbose = FALSE
  )

  solutions <- solve(problem)

  # Keep one representative for each distinct decision vector
  unique_decisions <- solution_unique(
    solutions,
    by = "decisions"
  )

  # Keep one representative for each distinct objective point
  unique_objectives <- solution_unique(
    solutions,
    by = "objectives"
  )

  # Compare only selected objectives
  unique_cost_benefit <- solution_unique(
    solutions,
    by = "objectives",
    objectives = c("cost", "benefit")
  )

  # Keep the last representative of each duplicate group
  unique_last <- solution_unique(
    solutions,
    by = "decisions",
    keep = "last"
  )

  # Compare the number of stored solutions
  sum(!is.na(get_runs(solutions)$solution_id))
  sum(!is.na(get_runs(unique_decisions)$solution_id))
  sum(!is.na(get_runs(unique_objectives)$solution_id))

  # Typical cleaning workflow
  if (requireNamespace("moocore", quietly = TRUE)) {
    clean_solutions <- solutions |>
      solution_filter(
```

```
      feasible_only = TRUE,
      nondominated = TRUE
    ) |>
    solution_unique(
      by = "decisions"
    )
  }
}
```

`solutionset-class` *SolutionSet class*

Description

The `SolutionSet` class stores the result of solving a `Problem` object.

A `SolutionSet` object is the standard result object returned by `solve`. It may contain a single solved run, as in ordinary single-objective optimization, or multiple solved runs, as in weighted-sum, ϵ-constraint, or AUGMECON workflows.

The object stores the original problem, run-level design and outcome tables, objective values, diagnostics, method metadata, and additional metadata needed to inspect, compare, and analyse optimization results.

Details

Conceptual role

The `SolutionSet` class represents the result of a solving workflow:

```
Problem -> solve() -> SolutionSet
```

Single-objective workflows are represented as `SolutionSet` objects with one run. Multi-objective workflows are represented as `SolutionSet` objects with one or more runs generated by the selected method.

This design provides a common result structure for downstream analysis, including objective-value extraction, run diagnostics, trade-off analysis, Pareto-frontier summaries, distance-based ranking, and comparison between scenarios.

Typical use cases

A `SolutionSet` is used to inspect and analyse the results returned by `solve`. Typical use cases include:

- inspecting selected planning units or actions;
- extracting run-level objective values;
- comparing alternative solutions produced by a multi-objective method;
- analysing trade-offs between objectives;
- filtering feasible, unique, or non-dominated solutions;

- computing distances to observed ideal or nadir points;
- preparing plots and tables for reporting.

Internal structure

A `SolutionSet` object separates information into several layers:

`problem` The `Problem` object used to generate the solution set.

`solution` A named `list` containing the core solving outputs. This typically includes the run design, the run summary table, and the stored run-level solution objects used internally by **multiscape**.

`summary` A named `list` containing user-facing summaries aggregated across runs when such summaries are available.

`diagnostics` A named `list` containing diagnostics about the solution set as a whole.

`method` A named `list` describing the optimization method used.

`meta` A named `list` containing additional metadata.

`name` A `character(1)` identifier for the solution set.

Run-level content

The `solution` field is the main internal container for run-level information. Typical entries include:

`design` A `data.frame` describing the optimization design. In multi-objective workflows this may include weights, ϵ-levels, or other method-specific parameters. In single-objective workflows this table usually contains one row.

`runs` A `data.frame` summarizing the outcome of each run. This typically includes run identifiers, solver status, runtime, optimality gap, objective values, and method-specific information.

`solutions` A list of stored run-level solution objects used internally by **multiscape**. Users should normally rely on accessor functions rather than directly manipulating this list.

In most workflows, the `runs` table is the most useful compact representation of the solution set. Detailed inspection should preferably be done through accessor functions such as `get_pu`, `get_actions`, `get_features`, and `get_targets`.

Objective values and run tables

The run table may store objective values in columns named `value_<alias>`, where `<alias>` is the alias of a registered objective. For example:

- `value_cost`,
- `value_frag`,
- `value_benefit`.

This convention provides a compact format for downstream functions that analyse trade-offs, frontiers, or distances between solutions.

Depending on the solving method, the run table may also contain design columns such as:

- `weight_<alias>` for weighted-sum runs;
- `eps_<alias>` for ϵ-constraint or AUGMECON runs.

Single-objective and multi-objective results

`solve()` always returns a `SolutionSet`. For a single-objective problem, the returned object contains one run. For a multi-objective workflow, the returned object contains one or more runs generated by the selected method.

This unified output structure makes it possible to use the same inspection and analysis functions regardless of whether the original problem was single-objective or multi-objective.

Diagnostics

The `diagnostics` field stores metadata about the solve process. Depending on the workflow, it may summarize:

- number of design rows;
- number of attempted runs;
- number of stored solutions;
- status frequencies;
- runtime ranges;
- gap ranges;
- and other aggregate information about the set of runs.

Printing

The `print()` method provides a concise summary of the solution set. It reports:

- the optimization method and participating objective aliases;
- the run-design type when this information is available;
- the number of design rows, attempted runs, stored solutions, and runs without a stored solution;
- run-level status, runtime, and optimality-gap summaries;
- the observed range of each objective across stored solutions;
- and any filtering, uniqueness, or append metadata already recorded in the object.

Printing does not calculate non-dominance, uniqueness, similarities, selection frequencies, or distances. These analyses are intentionally kept outside the print method so that printing remains fast and does not depend on optional packages.

This printed output is intended as a quick overview. Detailed inspection should use public accessor and analysis functions such as `get_runs`, `get_objectives`, `solution_filter`, and `frontier_distances`.

Value

No return value. This page documents the `SolutionSet` class.

Fields

`problem` The `Problem` object used to generate the solution set.

`solution` A named `list` containing the core solving outputs, typically including `design`, `runs`, and internally stored run-level solutions.

`summary` A named `list` containing user-facing summaries associated with the solution set.

`diagnostics` A named `list` containing diagnostics about the solution set as a whole.

`method` A named `list` describing the optimization method used.

`meta` A named `list` of additional metadata.

`name` A `character(1)` name for the solution set object.

Methods

`print()` Print a concise summary of the solution set, including method name, number of runs, and run-level diagnostics.

`show()` Alias of `print()`.

`repr()` Return a short one-line representation of the solution set.

`getMethod()` Return the method specification stored in `self$method`.

`getDesign()` Return the design table stored in `self$solution$design`.

`getRuns()` Return the run summary table stored in `self$solution$runs`.

`getSolutions()` Return the internally stored run-level solution objects. This is mainly intended for advanced use; most users should prefer higher-level accessor functions.

See Also

`solve`, `plot_tradeoff`, `get_pu`, `get_actions`, `get_features`, `get_targets`

`solve` *Solve a planning problem*

Description

Solve a planning problem stored in a `Problem` object.

This is the main execution step of the **multiscape** workflow. It reads the problem specification stored in a `Problem` object, builds the corresponding optimization model when needed, applies the configured solver settings, and returns a `solutionset-class` object.

A `SolutionSet` is the standard result object returned by `solve()`. Single-objective workflows are represented as one-run `SolutionSet` objects, while multi-objective workflows are represented as multi-run `SolutionSet` objects.

Usage

```
solve(x, ...)

## S3 method for class 'Problem'
solve(x, ...)
```

Arguments

x	A Problem object created with create_problem and optionally enriched with actions, effects, targets, constraints, objectives, spatial relations, method settings, and solver settings.
...	Additional arguments reserved for internal or legacy solver handling. These are not part of the main recommended user interface.

Details

Role of solve()

The typical **multiscape** workflow is:

```
x <- create_problem(...)
x <- add_...(x, ...)
x <- set_...(x, ...)
res <- solve(x)
```

Thus, solve() is the stage at which the stored problem specification is turned into one or more optimization runs.

For most users, solve() is the standard execution entry point. Explicit compilation with compile_model is optional and is mainly useful for advanced inspection or debugging workflows.

What solve() uses from the problem object

The function uses the information already stored in the Problem object, including:

- baseline planning data;
- actions, effects, profit, and spatial relations;
- targets and constraints;
- registered objectives;
- an optional multi-objective method configuration;
- solver settings.

If a model has not yet been built, it is built internally during the solve process. If a model snapshot or pointer already exists, the solving layer may reuse or refresh it depending on the internal model state.

Single-objective and multi-objective behaviour

solve() always returns a solutionset-class object.

- **Single-objective case.** If exactly one objective is active and no multi-objective method is configured, solve() runs a single optimization problem and returns a one-run SolutionSet.
- **Multi-objective case.** If a multi-objective method is configured, solve() dispatches internally according to the stored method name and returns a multi-run SolutionSet.

This unified output structure makes it possible to use the same inspection, plotting, and analysis functions regardless of whether the original problem was single-objective or multi-objective.

Currently supported multi-objective method names are:

- "weighted";
- "epsilon_constraint";
- "augmecon".

Consistency rule

If multiple objectives are registered but no multi-objective method has been selected, `solve()` stops with an error. In practical terms:

- one objective and no multi-objective method ⇒ single-objective solve;
- multiple objectives and a valid multi-objective method ⇒ multi-objective solve;
- multiple objectives and no multi-objective method ⇒ error.

Implicit conservation-planning model

If no explicit actions and no explicit effects are provided, `solve()` can build the corresponding classical conservation-planning formulation internally. In this case, selecting a planning unit is interpreted as applying an implicit conservation action, and the baseline feature amounts stored in `dist_features` count toward representation targets.

This allows standard reserve-selection problems to be solved without requiring users to manually define actions and effects. Explicit action-based workflows remain available by using `add_actions` and `add_effects`.

Solver settings

Solver configuration is read from the `Problem` object, typically after calling `set_solver` or one of its convenience wrappers such as `set_solver_gurobi` or `set_solver_cbc`.

These settings may include:

- the selected backend;
- time limits;
- optimality-gap settings;
- CPU cores;
- verbosity options;
- backend-specific solver parameters.

Method dispatch

`solve()` is an S3 generic. The public method documented here is `solve.Problem()`, which operates on `Problem` objects.

Value

A `solutionset-class` object.

The returned object contains run-level information, solver diagnostics, objective values, and stored optimization outputs. For single-objective problems, the returned `SolutionSet` contains one run. For multi-objective workflows, it contains one or more runs generated by the selected method.

Users will typically inspect or visualize results using accessor and plotting functions such as `get_pu`, `get_actions`, `get_features`, `get_targets`, and `plot_tradeoff`.

See Also

problem-class, solutionset-class, compile_model, set_solver, set_solver_cbc, set_solver_gurobi, set_method_weighted_sum, set_method_epsilon_constraint, set_method_augmecon, add_actions, add_effects

Examples

```
# Load a complete simulated planning problem.
example_data <- load_sim_multiaction()

x <- create_problem(
  pu = example_data$planning_units,
  features = example_data$features,
  dist_features = example_data$dist_features,
  cost = "cost"
) |>
  add_actions(
    example_data$actions,
    cost = example_data$action_costs
  ) |>
  add_effects(
    example_data$effect_assumptions
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost", include_pu_cost = FALSE)

if (requireNamespace("rcbc", quietly = TRUE)) {
  x <- set_solver_cbc(x, verbose = FALSE)
  solset <- solve(x)
  print(solset)
}
```

Index

∗ **Decision-space analysis**
selection_consistency, 145
∗ **Objective–decision linkage**
linkage_contrasts, 117
linkage_distances, 119
linkage_transition, 121
linkage_turnover, 124
∗ **Objective-space analysis**
frontier_neighbors, 100
∗ **SolutionSet inspection**
get_solution_states, 113
∗ **datasets**
sim_dist_features, 180
sim_features, 180
sim_multiaction, 180
sim_pu, 181
sim_pu_sf, 182

add_action_sets, 4, *7*, *9*, *13*, *44*, *144*
add_action_sets(), *17–19*, *22*, *102*
add_actions, 6, *11*, *13*, *25, 26*, *30*, *32*, *34*, *42*, *44*, *49*, *70, 71*, *90*, *142*, *144*, *196, 197*
add_actions(), *5*, *16*, *18*, *21*, *127*
add_benefits, 11, *49*
add_constraint_action_cardinality, 12, *144*
add_constraint_action_cardinality(), *5*, *16, 17*, *19*, *22*
add_constraint_action_excludes, 16
add_constraint_action_excludes(), *19*, *22*
add_constraint_action_requires, 18
add_constraint_action_requires(), *17*, *22*
add_constraint_action_together, 21
add_constraint_action_together(), *17*, *19*
add_constraint_area, 23
add_constraint_budget, *13*, 27
add_constraint_locked_actions, *9*, 30, *34*
add_constraint_locked_planning_units, 33, *35, 36*
add_constraint_locked_pu, *31, 32*, *34*, 35, *90*
add_constraint_targets_absolute, 36, *41*, *142*, *144*
add_constraint_targets_relative, *38*, 39, *142*
add_effects, *11, 12*, 42, *48, 49*, *51*, *67*, *69*, *71*, *142*, *144*, *181*, *196, 197*
add_effects(), *5*, *127*
add_losses, *12*, 48
add_objective_max_benefit, *12*, 49, *69*
add_objective_max_effect, 50, *67*
add_objective_max_effect(), *49*, *60*
add_objective_max_net_profit, 52, *55, 56*, *58*, *71*
add_objective_max_profit, *53*, 54, *58*, *71*
add_objective_min_cost, *53*, 56, *56*
add_objective_min_cost(), *127*
add_objective_min_effect, *51*, 59
add_objective_min_fragmentation_action, 61, *64*, *75*
add_objective_min_fragmentation_planning_units, *62*, 63, *65, 66*
add_objective_min_fragmentation_pu, *64*, 65, *75*
add_objective_min_intervention_impact, 66
add_objective_min_loss, *49*, *67*, 68
add_objective_min_loss(), *60*
add_profit, 70, *142*
add_spatial_boundary, *62*, *64*, 73, *80–82*, *84*, *90*
add_spatial_distance, 76, *79*, *81, 82*
add_spatial_knn, *76, 77*, 78, *81, 82*, *90*
add_spatial_queen, 80, *81, 82*, *84*
add_spatial_relations, *62*, *64*, *75*, *77*, *79*, 81

add_spatial_rook, *80–82*, 83

compile_model, 85, *195*, *197*
create_problem, *7*, *9*, *11*, *26*, *29*, *33, 34*, *36*, *42*, *49*, *70*, *76*, *78*, *80, 81*, *83*, 86, *142*, *144*, *195*
create_problem,ANY,ANY,missing-method *(*create_problem*)*, 86
create_problem,ANY,ANY,NULL-method *(*create_problem*)*, 86
create_problem,ANY,data.frame,data.frame-method *(*create_problem*)*, 86
create_problem,data.frame,data.frame,data.frame-method *(*create_problem*)*, 86

frontier_distances, 91, *99*, *101*, *108*, *120*, *193*
frontier_extremes, *93*, 95, *99*, *108*
frontier_knee, 97
frontier_neighbors, 100, *120*, *122*, *124, 125*

get_action_sets, 102
get_action_sets(), *5*
get_actions, 103, *106*, *110*, *114*, *116*, *130–132*, *148*, *151*, *186*, *192*, *194*, *196*
get_actions(), *102*
get_features, *104*, 105, *110*, *116*, *134*, *192*, *194*, *196*
get_objectives, *93*, *96*, *99*, 107, *112*, *183*, *186*, *189*, *193*
get_planning_units, *104*, *106*, 109, *111*, *114*, *116*, *135, 136*, *148*, *151*, *186*
get_pu, 111, *192*, *194*, *196*
get_runs, *108*, 112, *163, 164*, *166*, *183*, *186*, *193*
get_solution_states, 113, *122*, *146*
get_targets, *104*, *106*, *110*, 115, *192*, *194*, *196*

linkage_contrasts, 117, *120*, *122*, *125*
linkage_distances, *101*, *117, 118*, 119, *122*, *124, 125*
linkage_transition, *114*, *118*, *120*, 121, *125*
linkage_turnover, *101*, *117, 118*, *120*, *122*, 124
load_ecosystem_services, 126
load_ecosystem_services(), *128*, *182*
load_meseta, 127
load_meseta(), *180, 181*
load_sim_features_raster, 128
load_sim_multiaction, 128
load_sim_multiaction(), *180*

mo_control, 129

plot_spatial_actions, 130, *134*, *136*, *161*
plot_spatial_features, *131*, 132, *136*
plot_spatial_planning_units, *131*, *134*, 135, *136–138*
plot_spatial_pu, 137
plot_tradeoff, 138, *161*, *183*, *194*, *196*
Problem, *191*
Problem *(*problem-class*)*, 141
problem-class, 141

run_grid, 144
run_manual, 145

selection_consistency, *114*, 145, *148*, *151*
selection_frequency, *146*, 147, *151*
selection_similarity, *120*, *146*, *148*, 149
set_method_augmecon, *129, 130*, 152, *161, 162*, *164–166*, *168*, *197*
set_method_epsilon_constraint, 156, *161, 162*, *164–166*, *168*, *197*
set_method_weighted_sum, 159, *165, 166*, *168*, *197*
set_runs_control, *129, 130*, *161*, 162
set_runs_grid, *112*, *144*, *161*, 164, *164*, *167, 168*
set_runs_manual, *112*, *145*, *161*, *164*, *166*, 167
set_solver, 169, *173, 174*, *176–179*, *196, 197*
set_solver_cbc, *172*, 173, *196, 197*
set_solver_cplex, *172*, 174
set_solver_gurobi, *172*, 176, *196, 197*
set_solver_symphony, *172*, 178
sim_dist_features, 180
sim_features, 180
sim_multiaction, 180
sim_pu, 181
sim_pu_sf, 182
solution_append, 182, *189*
solution_filter, *93*, *96*, *98, 99*, *101*, *108*, *112*, *146*, *148*, *150, 151*, *183*, 184, *189*, *193*

solution_unique, *101*, *148*, *150, 151*, 188
SolutionSet *(*solutionset-class*)*, 191
solutionset-class, 191
solve, *85*, *90, 91*, *95*, *97*, *100*, *103*, *105*, *108–113*, *115*, *120*, *122*, *124*, *130–133*, *135–137*, *140, 141*, *144, 145*, *147*, *150*, *161*, *169–172*, *174*, *176, 177*, *179*, *182*, *185, 186*, *188*, *191*, 194, *194*