Package 'multiscape'

May 9, 2026

Type Package

Version 1.0.7

Title 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

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

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_actions.R' 'add_constraint_area.R'
'add_constraint_budget.R' 'add_constraint_locked_actions.R'
'add_constraint_locked_pu.R' 'add_constraint_targets.R'
'add_effects.R' 'add_objectives.R' 'add_profit.R'
'add_spatial_relations.R' 'build_model.R' 'compile_model.R'
'create_problem.R' 'data-sim.R' 'internalMO.R' 'get_solution.R'
'globals.R' 'load_sim.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_solver.R' 'solutionset-class.R' 'solution-class.R'
'show.R' 'solve.R' 'utils-pipe.R' 'zzz.R'

Config/testthat/edition 3

NeedsCompilation yes

Author José Salgado-Rojas [aut, cre] (ORCID:
<https://orcid.org/0000-0001-9817-2625>),
Matías Moreno-Faguett [aut] (ORCID:
<https://orcid.org/0009-0003-6561-234X>),
Núria Aquilué [aut] (ORCID: <https://orcid.org/0000-0001-7911-3144>)

Maintainer José Salgado-Rojas <jose.salgroj@gmail.com>

Repository CRAN

Date/Publication 2026-04-30 08:30:19 UTC

Contents

add_actions *Add management actions to a planning problem*

Description

Define the action catalogue, the set of feasible planning unit–action pairs, and their implementation costs.

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, 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
)
```

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`.

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.

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.

In the spatial case, feasible planning units are identified using `sf::st_intersects()` against the stored planning-unit geometry.

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.

Replacement behaviour.

Calling `add_actions()` replaces any previous action catalogue and feasible action table stored in the problem object.

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`, `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_constraint_locked_actions`

Examples

```
# ------------------------------------------------------
# Minimal planning problem
# ------------------------------------------------------
pu <- data.frame(
  id = 1:4,
  cost = c(2, 3, 1, 4)
)
```

```
features <- data.frame(
  id = 1:2,
  name = c("sp1", "sp2")
)

dist_features <- data.frame(
  pu = c(1, 1, 2, 3, 4, 4),
  feature = c(1, 2, 1, 2, 1, 2),
  amount = c(1, 2, 1, 3, 2, 1)
)

p <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features
)

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)
)

p3$data$dist_actions
```

`add_benefits` *Add benefits*

Description

Convenience wrapper around `add_effects` that keeps only positive effects, that is, rows with `benefit > 0`.

This function is useful when the workflow is focused only on beneficial consequences of actions and a gain-only wrapper is more convenient than calling `add_effects(..., component = "benefit")` directly. Internally, it calls `add_effects()` with `component = "benefit"`. The canonical stored result therefore contains only rows whose effect satisfies $\text{benefit}_{iaf} > 0$.

For backwards compatibility, a mirror table containing only the benefit component is also written to `x$data$dist_benefit`. The canonical filtered representation remains the one stored by `add_effects()`.

Usage

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

Arguments

`x`	A `Problem` object created with `create_problem`. It must already contain feasible actions; run `add_actions` first.
`benefits`	Alias of `effects`, kept for backwards compatibility.
`effect_type`	Character string indicating how supplied effect values are interpreted. Must be one of: • `"delta"`: values represent signed net changes, • `"after"`: values represent after-action amounts and are converted to net changes relative to baseline feature amounts.
`effect_aggregation`	Character string giving the aggregation used when converting raster values to planning-unit level. Must be one of `"sum"` or `"mean"`.

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`

Examples

```
pu <- data.frame(id = 1:2, cost = c(1, 2))
features <- data.frame(id = 1, name = "sp1")
dist_features <- data.frame(pu = 1:2, feature = 1, amount = c(5, 10))

p <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features
)

p <- add_actions(
  p,
  data.frame(id = "restoration")
)

eff <- data.frame(
  pu = c(1, 2),
  action = c("restoration", "restoration"),
  feature = c(1, 1),
  delta = c(2, -1)
)

p <- add_benefits(p, benefits = eff)
p$data$dist_benefit
```

---

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. 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 area column in x$data$pu. If NULL, the area source is resolved later by the model builder.
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 area in the problem through the planning-unit selection variables. Otherwise, it applies to the selected decision variables associated with the specified subset of actions. 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

Use this function when area requirements must be imposed either on the total selected landscape or on the subset of selected decisions associated with specific actions.

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`.

When `actions` is not `NULL`, the constraint is applied only to the selected decisions associated with the specified subset of actions. Let $\mathcal{A}^\star \subseteq \mathcal{A}$ denote that 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 $i \in \mathcal{I}$. In that case, the constrained quantity is

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

Under formulations where at most one action can be selected per planning unit, this coincides with the area allocated to that subset of actions.

Areas are obtained from the planning-unit table. If `area_col` is provided, that column is used. Otherwise, the model builder later determines the default area source according to the internal rules of the package. The value of `area_unit` indicates the unit in which `area` and `tolerance` are expressed and therefore how the stored threshold should be interpreted.

This function only stores the constraint specification; it does not validate the feasibility of the threshold against the available planning units 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`

Examples

```
pu <- data.frame(
  id = 1:4,
  cost = c(2, 3, 1, 4),
  area_ha = c(10, 15, 8, 20)
)
```

```
features <- data.frame(
  id = 1:2,
  name = c("sp1", "sp2")
)

dist_features <- data.frame(
  pu = c(1, 1, 2, 3, 4, 4),
  feature = c(1, 2, 1, 2, 1, 2),
  amount = c(1, 2, 1, 3, 2, 1)
)

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

p <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features
)

p <- add_actions(
  p,
  actions = actions,
  cost = c(conservation = 1, restoration = 2)
)

p <- add_constraint_area(
  x = p,
  area = 25,
  sense = "min",
  area_col = "area_ha",
  area_unit = "ha"
)

p <- add_constraint_area(
  x = p,
  area = 15,
  sense = "max",
  area_col = "area_ha",
  area_unit = "ha",
  actions = "restoration"
)

p <- add_constraint_area(
  x = p,
  area = 5,
  sense = "min",
  area_col = "area_ha",
  area_unit = "ha",
  actions = "restoration"
)
```

```
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

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

```
pu <- data.frame(
  id = 1:4,
  cost = c(2, 3, 1, 4)
)

features <- data.frame(
  id = 1:2,
  name = c("sp1", "sp2")
)

dist_features <- data.frame(
  pu = c(1, 1, 2, 3, 4, 4),
  feature = c(1, 2, 1, 2, 1, 2),
  amount = c(1, 2, 1, 3, 2, 1)
)

actions <- data.frame(
  id = c("conservation", "restoration")
)
```

```
p <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features
)

p <- add_actions(
  p,
  actions = actions,
  cost = c(conservation = 1, restoration = 2)
)

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 = "restoration",
  include_pu_cost = FALSE,
  include_action_cost = TRUE
)

p <- add_constraint_budget(
  x = p,
  budget = 1,
  sense = "min",
  actions = "restoration",
  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

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

```
pu <- data.frame(
  id = 1:4,
  cost = c(2, 3, 1, 4)
)

features <- data.frame(
  id = 1:2,
  name = c("sp1", "sp2")
)

dist_features <- data.frame(
  pu = c(1, 1, 2, 3, 4, 4),
  feature = c(1, 2, 1, 2, 1, 2),
  amount = c(1, 2, 1, 3, 2, 1)
)

p <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features
)

p <- add_actions(
  x = p,
  actions = data.frame(id = c("conservation", "restoration")),
  cost = c(conservation = 3, restoration = 8)
)
```

```
# Lock a few feasible decisions
p <- add_constraint_locked_actions(
  x = p,
  locked_in = data.frame(
    pu = c(1, 2),
    action = c("conservation", "restoration")
  ),
  locked_out = data.frame(
    pu = c(4),
    action = c("conservation")
  )
)

p$data$dist_actions

# Named-list interface
p2 <- add_constraint_locked_actions(
  x = p,
  locked_in = list(
    conservation = c(1, 3)
  ),
  locked_out = list(
    restoration = c(2)
  )
)

p2$data$dist_actions
```

`add_constraint_locked_pu` *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_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.

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.

Replacement behaviour

Each call to add_constraint_locked_pu() replaces any existing locked_in and locked_out columns in the planning-unit table. In other words, the function defines the complete current set of locked planning units; it does not merge new values with previous ones.

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`

Examples

```
pu <- data.frame(
  id = 1:5,
  cost = c(2, 3, 1, 4, 2),
  lock_col = c(TRUE, FALSE, FALSE, TRUE, FALSE),
  out_col = c(FALSE, FALSE, FALSE, FALSE, TRUE)
)

features <- data.frame(
  id = 1:2,
  name = c("sp1", "sp2")
)

dist_features <- data.frame(
  pu = c(1, 1, 2, 3, 4, 4),
  feature = c(1, 2, 1, 2, 1, 2),
  amount = c(1, 2, 1, 3, 2, 1)
)

p <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features
)

# 1) Lock by planning-unit ids
p1 <- add_constraint_locked_pu(
  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 PU data columns
p2 <- add_constraint_locked_pu(
  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_pu(
  x = p,
  locked_in = c(TRUE, FALSE, TRUE, FALSE, FALSE),
  locked_out = c(FALSE, FALSE, FALSE, TRUE, FALSE)
)

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

`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

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

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

When the optimization model is built, each such target is interpreted as a lower-bound constraint 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.

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

```
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)

feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)

dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)

p <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  cost = "cost"
) |>
  add_actions(data.frame(id = "conservation", name = "conservation"), cost = 0)

# 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 = "conservation"
)
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 current baseline 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 current baseline 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.

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 current full landscape baseline.

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 $[0, 1]$.

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

```
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)

feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)

dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)

p <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  cost = "cost"
) |>
  add_actions(data.frame(id = "conservation", name = "conservation"), cost = 0)

# 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 = "conservation"
)
p3$data$targets
```

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

Description

Define the effects of management actions on features across planning units.

Effects are stored in a canonical representation in a effects table, with one row per `(pu, action, feature)` triple and two non-negative columns:

- `benefit`: the positive component of the effect,
- `loss`: the magnitude of the negative component of the effect.

The net effect is therefore interpreted as

$$\Delta_{iaf} = \text{benefit}_{iaf} - \text{loss}_{iaf},$$

where i indexes planning units, a indexes actions, and f indexes features.

Under the semantics adopted by this package, each `(pu, action, feature)` triple represents a single net effect. Consequently, after validation and aggregation, a stored row cannot have both `benefit > 0` and `loss > 0` at the same time.

Usage

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

Arguments

`x`	A `Problem` object created with `create_problem`. It must already contain feasible actions; run `add_actions` first.
`effects`	Effect specification. One of: • `NULL`, to store an empty effects table, • a `data.frame(action, feature, multiplier)`, • a `data.frame(pu, action, feature, ...)` with explicit effects, • a named list of `terra::SpatRaster` objects, one per action.
`effect_type`	Character string indicating how supplied effect values are interpreted. Must be one of: • `"delta"`: values represent signed net changes, • `"after"`: values represent after-action amounts and are converted to net changes relative to baseline feature amounts.
`effect_aggregation`	Character string giving the aggregation used when converting raster values to planning-unit level. Must be one of `"sum"` or `"mean"`.
`component`	Character string controlling which component of the canonical effects table is retained. Must be one of:

- "any": keep all non-zero rows,
- "benefit": keep only rows with benefit > 0,
- "loss": keep only rows with loss > 0.

Details

When to use add_effects().

Use this function when you want to specify what feasible actions do to features. It is the stage at which an action-based decision space is linked to feature-level ecological or functional consequences.

This function provides a unified interface for specifying action effects from several input formats while enforcing a single internal representation. Regardless of how the user supplies the effects, the stored output always follows the same canonical structure based on non-negative benefit/loss components.

Let $i \in \mathcal{I}$ index planning units, $a \in \mathcal{A}$ index actions, and $f \in \mathcal{F}$ index features. Let b_{if} denote the baseline amount of feature f in planning unit i, as given by the feature-distribution table. Let Δ_{iaf} denote the net change caused by applying action a in planning unit i to feature f. The canonical stored representation is:

$$\text{benefit}_{iaf} = \max(\Delta_{iaf}, 0),$$

$$\text{loss}_{iaf} = \max(-\Delta_{iaf}, 0).$$

Hence:

- if $\Delta_{iaf} > 0$, then benefit > 0 and loss = 0,
- if $\Delta_{iaf} < 0$, then benefit = 0 and loss > 0,
- if $\Delta_{iaf} = 0$, then both are zero.

Why split effects into benefit and loss?

This representation avoids ambiguity in downstream optimization models. It allows the package to support, for example, objectives that maximize beneficial effects, minimize damages, impose no-net-loss conditions, or combine both components differently in multi-objective formulations.

Supported effect specifications

The effects argument may be provided in one of the following forms:

1. NULL. An empty effects table is stored.
2. A data.frame(action, feature, multiplier). In this case, effects are constructed by multiplying baseline feature amounts by the supplied multiplier:

   $$\Delta_{iaf} = b_{if} \times m_{af},$$

   where m_{af} is the multiplier associated with action a and feature f. This specification is expanded over all feasible (pu, action) pairs.
3. A data.frame(pu, action, feature, ...) giving explicit effects for individual triples. The table may contain:

- `delta` or `effect`: interpreted as signed net changes,
- `after`: interpreted as after-action amounts if `effect_type = "after"`,
- `benefit` and/or `loss`: explicit non-negative split components,
- legacy signed `benefit` without `loss`: interpreted as a signed net effect for backwards compatibility.

4. A named list of `terra::SpatRaster` objects, one per action. In this case, names must match action ids, and each raster must contain one layer per feature. Raster values are aggregated to planning-unit level using `effect_aggregation`.

Interpretation of `effect_type`

If `effect_type = "delta"`, supplied values are interpreted as net changes directly.

If `effect_type = "after"`, supplied values are interpreted as after-action amounts and converted internally to net effects using:

$$\Delta_{iaf} = \text{after}_{iaf} - b_{if}.$$

Missing baseline values are treated as zero.

Feasibility and locked-out decisions

Effects are only retained for feasible `(pu, action)` pairs. Thus, `add_actions()` must be called first. Pairs marked as locked out (`status == 3`) are removed before storing the final effects table.

This function does not define the action-decision layer itself; it builds on the feasible `(pu, action)` pairs already stored in the problem.

Duplicate rows and semantic validation

If multiple rows are supplied for the same `(pu, action, feature)` triple, they are aggregated by summing `benefit` and `loss` separately. The resulting triple must still respect the package semantics, namely that both components cannot be strictly positive simultaneously. Inputs violating this rule are rejected.

Component filtering

After canonicalization and validation, rows can be restricted to:

- `component = "any"`: keep all non-zero effects,
- `component = "benefit"`: keep only rows with `benefit > 0`,
- `component = "loss"`: keep only rows with `loss > 0`.

Rows with `benefit = 0` and `loss = 0` are always removed.

Raster handling

When effects are supplied as rasters, they are automatically aligned to the planning-unit raster or geometry when needed before extraction or zonal aggregation.

Stored output

The resulting effects table contains user-facing ids, internal integer ids, and optional labels for actions and features. Metadata describing the stored representation and input interpretation are written to an effects metadata field.

After defining effects, typical next steps include adding objectives that use beneficial or harmful effects, and then solving the configured problem.

Value

An updated `Problem` object containing:

`dist_effects` A canonical effects table with columns `pu`, `action`, `feature`, `benefit`, `loss`, `internal_pu`, `internal_action`, `internal_feature`, and optional labels such as `feature_name` and `action_name`.

`effects_meta` Metadata describing how effects were interpreted and stored.

See Also

`add_actions`, `add_benefits`, `add_losses`

Examples

```
# ------------------------------------------------------
# Minimal problem with actions
# ------------------------------------------------------
pu <- data.frame(
  id = 1:3,
  cost = c(1, 2, 3)
)

features <- data.frame(
  id = 1:2,
  name = c("sp1", "sp2")
)

dist_features <- data.frame(
  pu = c(1, 1, 2, 3),
  feature = c(1, 2, 1, 2),
  amount = c(10, 5, 8, 4)
)

p <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features
)

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

p <- add_actions(
  x = p,
  actions = actions,
  cost = c(conservation = 2, restoration = 4)
)

print(p)
```

```
# ------------------------------------------------------
# 1) Empty effects
# ------------------------------------------------------
p0 <- add_effects(p, effects = NULL)
p0$data$dist_effects

# ------------------------------------------------------
# 2) Multipliers by action and feature
# delta = baseline amount * multiplier
# ------------------------------------------------------
mult <- data.frame(
  action = c("conservation", "conservation",
             "restoration", "restoration"),
  feature = c("sp1", "sp2", "sp1", "sp2"),
  multiplier = c(0.10, 0.00, -0.20, 0.25)
)

p1 <- add_effects(
  x = p,
  effects = mult,
  effect_type = "delta"
)

p1$data$dist_effects

# ------------------------------------------------------
# 3) Explicit net effects using signed deltas
# ------------------------------------------------------
eff_delta <- data.frame(
  pu = c(1, 2, 3, 3),
  action = c("conservation", "restoration", "restoration", "conservation"),
  feature = c(1, 1, 2, 2),
  delta = c(2, -3, 1, 0.5)
)

p2 <- add_effects(
  x = p,
  effects = eff_delta
)

p2$data$dist_effects

# ------------------------------------------------------
# 4) Explicit after-action amounts
# delta = after - baseline
# ------------------------------------------------------
eff_after <- data.frame(
  pu = c(1, 2, 3),
  action = c("conservation", "restoration", "restoration"),
  feature = c(1, 1, 2),
  after = c(12, 5, 6)
)
```

```
p3 <- add_effects(
  x = p,
  effects = eff_after,
  effect_type = "after"
)

p3$data$dist_effects

# ------------------------------------------------------
# 5) Keep only beneficial effects
# ------------------------------------------------------
p4 <- add_effects(
  x = p,
  effects = eff_delta,
  component = "benefit"
)

p4$data$dist_effects
```

`add_losses` *Add losses*

Description

Convenience wrapper around `add_effects` that keeps only negative effects, represented by rows with `loss > 0`.

This function is useful when the workflow is focused only on damaging consequences of actions and a loss-only wrapper is more convenient than calling `add_effects(..., component = "loss")` directly. Internally, it calls `add_effects()` with `component = "loss"`. The canonical stored result therefore contains only rows whose effect satisfies $\text{loss}_{iaf} > 0$.

In addition, a mirror table containing only the loss component is stored in `x$data$dist_loss`.

Usage

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

Arguments

`x`	A `Problem` object created with `create_problem`. It must already contain feasible actions; run `add_actions` first.
`losses`	Alias of `effects`, used for symmetry with `add_benefits()`.

`effect_type` Character string indicating how supplied effect values are interpreted. Must be one of:

- `"delta"`: values represent signed net changes,
- `"after"`: values represent after-action amounts and are converted to net changes relative to baseline feature amounts.

`effect_aggregation` Character string giving the aggregation used when converting raster values to planning-unit level. Must be one of `"sum"` or `"mean"`.

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`

Examples

```
pu <- data.frame(id = 1:2, cost = c(1, 2))
features <- data.frame(id = 1, name = "sp1")
dist_features <- data.frame(pu = 1:2, feature = 1, amount = c(5, 10))

p <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features
)

p <- add_actions(
  p,
  data.frame(id = "harvest")
)

eff <- data.frame(
  pu = c(1, 2),
  action = c("harvest", "harvest"),
  feature = c(1, 1),
  delta = c(2, -4)
)

p <- add_losses(p, losses = eff)
p$data$dist_loss
```

`add_objective_max_benefit` *Add objective: maximize benefit*

Description

Define an objective that maximizes the total positive effects generated by selected actions on selected features.

This objective is based on the canonical effects table and uses only the non-negative `benefit` component.

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.

Details

Use this function when positive ecological gains should be maximized explicitly, without offsetting them against harmful effects.

Let $b_{iaf} \geq 0$ denote the stored benefit associated with planning unit i, action a, and feature f. Since the effects table is already expressed in canonical form, b_{iaf} represents the positive part of the net effect associated with the corresponding selected action decision.

If no subsets are supplied, the objective can be written as:

$$\max \sum_{(i,a,f) \in \mathcal{R}} b_{iaf}\, x_{ia},$$

where $\mathcal{R}$ denotes the set of stored benefit rows and $x_{ia} \in \{0, 1\}$ indicates whether action a is selected in planning unit i.

If `actions` is provided, only rows whose action belongs to the selected subset contribute to the objective.

If `features` is provided, only rows whose feature belongs to the selected subset contribute to the objective.

More generally, letting $\mathcal{R}^\star$ be the subset induced by the selected actions and features, the objective is:

$$\max \sum_{(i,a,f) \in \mathcal{R}^{\star}} b_{iaf}\, x_{ia}.$$

This objective maximizes gains only. It does not subtract losses. If harmful effects should also be accounted for, they must be handled separately through additional objectives or constraints.

Value

An updated `Problem` object.

See Also

`add_objective_min_loss`, `add_effects`

Examples

```
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)
feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)
dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)
actions_df <- data.frame(
  id = c("conservation", "restoration"),
  name = c("conservation", "restoration")
)
effects_df <- data.frame(
  pu = c(1, 2, 3, 4, 1, 2, 3, 4),
  action = c("conservation", "conservation", "conservation", "conservation",
             "restoration", "restoration", "restoration", "restoration"),
  feature = c(1, 1, 1, 1, 2, 2, 2, 2),
  benefit = c(2, 1, 0, 1, 3, 0, 1, 2),
  loss = c(0, 0, 1, 0, 0, 1, 0, 0)
)

p <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  cost = "cost"
) |>
  add_actions(actions_df, cost = c(conservation = 1, restoration = 2)) |>
  add_effects(effects_df)
```

```
p1 <- add_objective_max_benefit(p)
p1$data$model_args

p2 <- add_objective_max_benefit(
  p,
  actions = "restoration"
)
p2$data$model_args

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

`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.

See Also

`add_objective_max_profit`, `add_objective_min_cost`

Examples

```
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)
feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)
dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)
actions_df <- data.frame(
```

```
  id = c("conservation", "restoration"),
  name = c("conservation", "restoration")
)
profit_df <- data.frame(
  pu = c(1, 2, 3, 4, 1, 2, 3, 4),
  action = c("conservation", "conservation", "conservation", "conservation",
             "restoration", "restoration", "restoration", "restoration"),
  profit = c(5, 4, 3, 2, 8, 7, 6, 5)
)

p <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  cost = "cost"
) |>
  add_actions(actions_df, cost = c(conservation = 1, restoration = 2)) |>
  add_profit(profit_df)

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

p2 <- add_objective_max_net_profit(
  p,
  include_pu_cost = FALSE,
  include_action_cost = TRUE
)
p2$data$model_args

p3 <- add_objective_max_net_profit(
  p,
  actions = "restoration"
)
p3$data$model_args
```

`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.

See Also

add_objective_min_cost, add_objective_max_net_profit

Examples

```
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)
feat_tbl <- data.frame(
  id = 1:2,
```

```
  name = c("feature_1", "feature_2")
)
dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)
actions_df <- data.frame(
  id = c("conservation", "restoration"),
  name = c("conservation", "restoration")
)
profit_df <- data.frame(
  pu = c(1, 2, 3, 4, 1, 2, 3, 4),
  action = c("conservation", "conservation", "conservation", "conservation",
             "restoration", "restoration", "restoration", "restoration"),
  profit = c(5, 4, 3, 2, 8, 7, 6, 5)
)

p <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  cost = "cost"
) |>
  add_actions(actions_df, cost = c(conservation = 1, restoration = 2)) |>
  add_profit(profit_df)

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

p2 <- add_objective_max_profit(
  p,
  actions = "restoration"
)
p2$data$model_args
```

`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.

See Also

`add_objective_max_profit`, `add_objective_max_net_profit`

Examples

```
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)
feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)
dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)
actions_df <- data.frame(
  id = c("conservation", "restoration"),
  name = c("conservation", "restoration")
)

p <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  cost = "cost"
) |>
  add_actions(actions_df, cost = c(conservation = 1, restoration = 2))

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

p2 <- add_objective_min_cost(
  p,
  include_pu_cost = FALSE,
  include_action_cost = TRUE
)
p2$data$model_args

p3 <- add_objective_min_cost(
  p,
  actions = "restoration"
)
p3$data$model_args
```

`add_objective_min_fragmentation_action` *Add objective: minimize action fragmentation*

Description

Define an objective that minimizes fragmentation at the action level over a stored spatial relation.

Unlike `add_objective_min_fragmentation_pu`, which acts on the selected planning-unit set through w_i, this objective acts on the spatial arrangement of individual action decisions through the action variables x_{ia}.

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

Use this function when spatial cohesion should be encouraged separately for each selected action pattern.

Let $\mathcal{I}$ denote the set of planning units and let $\mathcal{A}$ denote the set of actions.

Let $x_{ia} \in \{0, 1\}$ indicate whether action $a \in \mathcal{A}$ is selected in planning unit $i \in \mathcal{I}$.

Let the chosen spatial relation define weighted pairs with weights $\omega_{ij} \geq 0$, and let $\lambda =$ `weight_multiplier` be the global scaling factor applied to these weights.

If `actions` is supplied, only the selected subset $\mathcal{A}^\star \subseteq \mathcal{A}$ contributes to the final objective. If `actions = NULL`, all actions are included.

The internal preparation step constructs one auxiliary variable $b_{ija} \in [0, 1]$ for each unique non-diagonal undirected edge (i, j) with $i < j$ and for each action a. The intended semantics is:

$$b_{ija} = x_{ia} \wedge x_{ja}.$$

Whenever both decision variables x_{ia} and x_{ja} exist in the model, this conjunction is enforced by the linearization:

$$b_{ija} \leq x_{ia},$$

$$b_{ija} \leq x_{ja},$$

$$b_{ija} \geq x_{ia} + x_{ja} - 1.$$

If one of the two action variables does not exist because the corresponding `(pu, action)` pair is not feasible, the auxiliary variable is forced to zero.

Therefore, $b_{ija} = 1$ if and only if action a is selected in both adjacent planning units i and j; otherwise $b_{ija} = 0$.

The exact objective coefficients are assembled later by the model builder from:

- the action decision variables x_{ia},
- the edge-conjunction variables b_{ija},
- the relation weights ω_{ij},
- the multiplier λ,
- and, if supplied, the action-specific weights.

If action-specific weights are provided, let $\alpha_a \geq 0$ denote the weight associated with action a. Then the resulting objective can be interpreted as an action-wise compactness or fragmentation functional of the form:

$$\min \sum_{a \in \mathcal{A}^\star} \alpha_a \, F_a(x_{\cdot a}, b_{\cdot\cdot a}; \lambda\omega),$$

where F_a is the fragmentation expression induced by the selected relation and the internal coefficient construction for action a.

In practical terms, this objective penalizes solutions in which the same action is spatially scattered or broken into separate patches, while allowing different actions to form different spatial patterns.

This differs from planning-unit fragmentation:

- `add_objective_min_fragmentation_pu()` encourages cohesion of the union of selected planning units,
- `add_objective_min_fragmentation_action()` encourages cohesion of each selected action pattern separately.

Setting `weight_multiplier = 0` removes the contribution of the spatial relation from the objective after scaling.

Value

An updated `Problem` object.

See Also

add_objective_min_fragmentation_pu, add_spatial_boundary, add_spatial_relations

Examples

```
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)

feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)

dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)

actions_df <- data.frame(
  id = c("conservation", "restoration"),
  name = c("conservation", "restoration")
)

bound_df <- data.frame(
  id1 = c(1, 1, 2, 1, 2, 3, 4),
  id2 = c(1, 2, 2, 3, 4, 4, 4),
  boundary = c(4, 1, 4, 1, 1, 1, 4)
)

p <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  cost = "cost"
) |>
  add_actions(actions_df, cost = c(conservation = 1, restoration = 2))

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

p <- add_objective_min_fragmentation_action(
  p,
  relation_name = "boundary",
  actions = "restoration",
```

```
  weight_multiplier = 1
)

p$data$model_args
```

`add_objective_min_fragmentation_pu` *Add objective: minimize fragmentation*

Description

Define an objective that minimizes planning-unit fragmentation over a stored spatial relation.

This objective acts on the planning-unit selection pattern through the binary planning-unit variables w_i. It is therefore appropriate when spatial cohesion is to be encouraged at the level of the selected planning-unit set as a whole.

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.

Details

Use this function when spatial cohesion should be encouraged at the level of the selected planning-unit set as a whole.

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

Let the chosen spatial relation define a set of weighted pairs with weights $\omega_{ij} \geq 0$. These relation weights are interpreted by the model builder after scaling by $\lambda =$ `weight_multiplier`.

The internal preparation step constructs one auxiliary variable $y_{ij} \in [0, 1]$ for each unique non-diagonal undirected edge (i, j) with $i < j$. The intended semantics is:

$$y_{ij} = w_i \wedge w_j.$$

This is enforced by the standard linearization:

$$y_{ij} \leq w_i,$$

$$y_{ij} \leq w_j,$$

$$y_{ij} \geq w_i + w_j - 1.$$

Thus, $y_{ij} = 1$ if and only if both planning units i and j are selected, and $y_{ij} = 0$ otherwise.

The exact objective coefficients are assembled later by the model builder from:

- the planning-unit variables w_i,
- the edge-conjunction variables y_{ij},
- the stored relation weights ω_{ij},
- and the multiplier λ.

Conceptually, the resulting objective is a boundary- or relation-based compactness functional that penalizes exposed or weakly connected selected patterns while rewarding adjacency among selected planning units.

In the common case where `relation_name = "boundary"` and the relation was built with `add_spatial_boundary`, the objective corresponds to a boundary-length-style fragmentation penalty.

Setting `weight_multiplier = 0` removes the contribution of the spatial relation from the objective after scaling.

This objective does not distinguish between different actions within the same planning unit. If action-specific spatial cohesion is required, use `add_objective_min_fragmentation_action` instead.

Value

An updated `Problem` object.

See Also

`add_spatial_boundary`, `add_spatial_relations`, `add_objective_min_fragmentation_action`

Examples

```
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)

feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
```

```
)

dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)

actions_df <- data.frame(
  id = c("conservation", "restoration"),
  name = c("conservation", "restoration")
)

bound_df <- data.frame(
  id1 = c(1, 1, 2, 1, 2, 3, 4),
  id2 = c(1, 2, 2, 3, 4, 4, 4),
  boundary = c(4, 1, 4, 1, 1, 1, 4)
)

p <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  cost = "cost"
) |>
  add_actions(actions_df, cost = c(conservation = 1, restoration = 2))

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

p <- add_objective_min_fragmentation_pu(
  p,
  relation_name = "boundary"
)

p$data$model_args
```

`add_objective_min_intervention_impact` *Add objective: minimize intervention impact*

Description

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

Use this function when intervention itself has a baseline ecological, social, or operational burden that should be minimized independently of the detailed effects of particular actions.

Let $w_i \in \{0, 1\}$ denote whether planning unit i is selected for intervention. Let q_{if} denote the impact amount associated with planning unit i and feature f, taken from column impact_col in the feature-distribution table.

If all selected features are included, the objective can be interpreted as:

$$\min \sum_{i \in \mathcal{I}} \left(\sum_{f \in \mathcal{F}^\star} q_{if} \right) w_i,$$

where $\mathcal{F}^\star$ denotes the selected subset of features.

Thus, the coefficient attached to w_i is the aggregated impact of the selected features in planning unit i.

The role of actions in this objective is not to make impact additive over actions, but to restrict the notion of intervention to planning units that are relevant for the selected subset of actions in downstream model construction. Even when multiple feasible actions exist in a planning unit, the planning unit contributes at most once through w_i.

This objective is useful when intervention itself has a baseline ecological, social, or operational impact that should be minimized independently of the detailed gain or loss generated by particular actions.

Value

An updated `Problem` object.

See Also

`add_objective_max_benefit`, `add_objective_min_loss`

Examples

```
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)
feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)
dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)
actions_df <- data.frame(
  id = c("conservation", "restoration"),
  name = c("conservation", "restoration")
)

p <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  cost = "cost"
) |>
  add_actions(actions_df, cost = c(conservation = 1, restoration = 2))

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

p2 <- add_objective_min_intervention_impact(
  p,
  features = 1
)
p2$data$model_args

p3 <- add_objective_min_intervention_impact(
  p,
  actions = "restoration"
)
p3$data$model_args
```

`add_objective_min_loss` *Add objective: minimize loss*

Description

Define an objective that minimizes the total negative effects generated by selected actions on selected features.

This objective is based on the canonical effects table and uses only the non-negative `loss` component.

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

Use this function when harmful ecological effects should be minimized explicitly, without offsetting them against beneficial effects.

Let $\ell_{iaf} \geq 0$ denote the stored loss associated with planning unit i, action a, and feature f.

If no subsets are supplied, the objective can be written as:

$$\min \sum_{(i,a,f) \in \mathcal{R}} \ell_{iaf} \, x_{ia}.$$

where $\mathcal{R}$ denotes the set of stored loss rows and $x_{ia} \in \{0, 1\}$ indicates whether action a is selected in planning unit i.

If `actions` is provided, only rows whose action belongs to the selected subset contribute to the objective.

If `features` is provided, only rows whose feature belongs to the selected subset contribute to the objective.

More generally, letting $\mathcal{R}^\star$ be the subset induced by the selected actions and features, the objective is:

$$\min \sum_{(i,a,f)\in\mathcal{R}^\star} \ell_{iaf}\, x_{ia}.$$

This objective minimizes harmful effects only. It does not offset losses against benefits unless benefits are handled elsewhere through additional objectives or constraints.

Value

An updated `Problem` object.

See Also

`add_objective_max_benefit`, `add_effects`

Examples

```
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)
feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)
dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)
actions_df <- data.frame(
  id = c("conservation", "restoration"),
  name = c("conservation", "restoration")
)
effects_df <- data.frame(
  pu = c(1, 2, 3, 4, 1, 2, 3, 4),
  action = c("conservation", "conservation", "conservation", "conservation",
             "restoration", "restoration", "restoration", "restoration"),
  feature = c(1, 1, 1, 1, 2, 2, 2, 2),
  benefit = c(2, 1, 0, 1, 3, 0, 1, 2),
  loss = c(0, 0, 1, 0, 0, 1, 0, 0)
)

p <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  cost = "cost"
) |>
  add_actions(actions_df, cost = c(conservation = 1, restoration = 2)) |>
  add_effects(effects_df)
```

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

p2 <- add_objective_min_loss(
  p,
  actions = "restoration"
)
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

Define economic profit values for feasible planning unit–action pairs and store them in a profit table.

Profit is stored separately from ecological effects. In particular, `profit` is not the same as ecological `benefit` or `loss` as represented in `add_effects`. This separation allows the package to distinguish economic returns from ecological consequences when building objectives, constraints, and reporting summaries.

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.

Details

When to use `add_profit()`.

Use this function when economic returns, penalties, or other action-specific financial values are part of the planning problem. Typical downstream uses include objectives such as `add_objective_max_profit` and `add_objective_max_net_profit`.

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 currently stored in the problem.

This function assigns to each feasible pair $(i, a) \in \mathcal{D}$ a numeric profit value $\pi_{ia} \in \mathbb{R}$ and stores the result in a profit table.

Thus, the stored table can be interpreted as a mapping

$$\pi : \mathcal{D} \to \mathbb{R},$$

where π_{ia} represents the economic return associated with selecting action a in planning unit i.

Profit values may be positive, zero, or negative. Positive values represent gains or revenues, zero represents no net profit contribution, and negative values can be used to encode penalties or net economic losses.

The stored table contains:

- `pu`: external planning-unit id,
- `action`: action id,
- `profit`: numeric profit value,
- `internal_pu`: internal planning-unit index,
- `internal_action`: internal action index.

Supported input formats

The `profit` argument may be specified in several ways:

- `NULL`: assign profit 0 to all feasible `(pu, action)` pairs,
- a numeric scalar: assign the same profit value to all feasible pairs,
- a named numeric vector: names are action ids, assigning one global profit value per action,
- a `data.frame(action, profit)`: assign one global profit value per action,
- a `data.frame(pu, action, profit)`: assign pair-specific profit values.

When action-level profit is supplied, the same profit value is assigned to all feasible planning units for that action. When pair-specific profit is supplied, only the listed `(pu, action)` pairs receive explicit values; unmatched feasible pairs are interpreted as zero-profit pairs.

Storage behaviour

This function stores only rows with non-zero profit values. Feasible pairs whose final profit is zero are omitted from the stored profit table. Missing values produced during matching or joins are treated as zero before this filtering step. Therefore, the resulting table is a sparse representation of economic returns over the feasible decision space.

Data-only behaviour

This function is purely data-oriented. It does not build or modify the optimization model, and it does not change feasibility. It simply assigns profit values to rows already present in the feasible action table.

In particular:

- it does not add new feasible `(pu, action)` pairs,
- it does not remove infeasible pairs,
- it does not apply solver-side filtering such as dropping locked-out decisions,
- it does not modify ecological effect tables.

Any such filtering is expected to occur later when model-ready tables are prepared, typically during the build stage invoked by `solve()`.

Use in optimization

Profit values stored by this function can later be used in objectives such as `add_objective_max_profit` or `add_objective_max_net_profit`, in derived budget expressions, or in reporting and summary functions.

For example, if $x_{ia} \in \{0, 1\}$ denotes whether action a is selected in planning unit i, then a profit-maximization objective typically takes the form

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

Value

An updated `Problem` object with a stored profit table created or replaced. 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

```
# Minimal problem
pu <- data.frame(
  id = 1:4,
  cost = c(2, 3, 1, 4)
)

features <- data.frame(
  id = 1:2,
  name = c("sp1", "sp2")
)

dist_features <- data.frame(
  pu = c(1, 1, 2, 3, 4, 4),
  feature = c(1, 2, 1, 2, 1, 2),
  amount = c(1, 2, 1, 3, 2, 1)
```

```
)

p <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features
)

p <- add_actions(
  x = p,
  actions = data.frame(id = c("harvest", "restoration"))
)

# 1) Constant profit for every feasible (pu, action)
p1 <- add_profit(p, profit = 10)
p1$data$dist_profit

# 2) Profit per action using a named vector
pr <- c(harvest = 50, restoration = -5)
p2 <- add_profit(p, profit = pr)
p2$data$dist_profit

# 3) Profit per action using a data frame
pr_df <- data.frame(
  action = c("harvest", "restoration"),
  profit = c(40, 15)
)
p3 <- add_profit(p, profit = pr_df)
p3$data$dist_profit

# 4) Profit per (pu, action) pair
pr_pair <- data.frame(
  pu = c(1, 2, 3),
  action = c("harvest", "harvest", "restoration"),
  profit = c(100, 80, 30)
)
p4 <- add_profit(p, profit = pr_pair)
p4$data$dist_profit
```

`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

```
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)

feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)

dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
```

```
  amount = c(5, 2, 3, 4, 1)
)

bound_df <- data.frame(
  id1 = c(1, 1, 2, 1, 2, 3, 4),
  id2 = c(1, 2, 2, 3, 4, 4, 4),
  boundary = c(4, 1, 4, 1, 1, 1, 4)
)

p <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  cost = "cost"
)

p <- add_spatial_boundary(
  x = p,
  boundary = bound_df,
  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

```
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4),
  x = c(0, 1, 0, 1),
  y = c(0, 0, 1, 1)
)

feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)

dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)

p <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  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$x and x$datapuy.
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

```
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4),
  x = c(0, 1, 0, 1),
  y = c(0, 0, 1, 1)
)

feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)

dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)

p <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  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

```
library(terra)

data("sim_pu_sf", package = "multiscape")
sim_features <- load_sim_features_raster()

p <- create_problem(
  pu = sim_pu_sf,
  features = sim_features,
  cost = "cost"
)

p <- add_spatial_queen(
  x = p,
  geometry = sim_pu_sf,
  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 edges as undirected and collapse duplicate unordered pairs. If TRUE, keep edges as directed ordered pairs.
allow_self	Logical. If TRUE, allow self-edges (i, i). 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, duplicated undirected edges are collapsed automatically using the maximum weight observed for each unordered pair.

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.

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

If a relation with the same name already exists, it is replaced.

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

```
pu <- data.frame(id = 1:3, cost = c(1, 2, 3))

features <- data.frame(
  id = 1,
  name = "sp1"
)

dist_features <- data.frame(
  pu = 1:3,
  feature = 1,
```

```
  amount = c(1, 1, 1)
)

p <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features
)

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

```
library(terra)

data("sim_pu_sf", package = "multiscape")
sim_features <- load_sim_features_raster()

p <- create_problem(
  pu = sim_pu_sf,
  features = sim_features,
  cost = "cost"
)

p <- add_spatial_rook(
  x = p,
  geometry = sim_pu_sf,
  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

```
## Not run:
x <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features
)

x <- x |>
  add_constraint_targets_relative(0.3) |>
  add_objective_min_cost(alias = "cost")

x <- compile_model(x)

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

## End(Not run)
```

`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

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

```
# ------------------------------------------------------
# 1) Tabular mode
# ------------------------------------------------------
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)

feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)
```

```
dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)

p1 <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl
)

print(p1)

# -----------------------------------------------------
# 2) Hybrid sf + tabular mode using package data
# -----------------------------------------------------

p2 <- create_problem(
  pu = sim_pu,
  features = sim_features,
  dist_features = sim_dist_features,
  cost = "cost"
)

print(p2)
```

`get_actions` *Get action results from a solution*

Description

Extract the action-allocation summary table from a `Solution` or `SolutionSet` 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 the solution.

Usage

```
get_actions(x, only_selected = FALSE, run = NULL)
```

Arguments

`x`	A `Solution` or `SolutionSet` object returned by `solve`.
`only_selected`	Logical. If `TRUE`, return only rows where `selected == 1`. Default is `FALSE`.
`run`	Optional positive integer giving the run index to extract from a `SolutionSet`. If `NULL`, all runs are returned when available.

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 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 `x` is a `SolutionSet` and `run` is provided, only rows belonging to that run are returned. This requires the summary table to contain a `run_id` column.

If `only_selected = TRUE`, only rows with `selected == 1` are returned. This requires the summary table to contain a `selected` column.

This function is intended for user-facing inspection of action allocations. For the raw model variable vector, use `get_solution_vector`.

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_pu`, `get_features`, `get_targets`, `get_solution_vector`

Examples

```
if (requireNamespace("rcbc", quietly = TRUE)) {
  pu_tbl <- data.frame(
    id = 1:4,
    cost = c(1, 2, 3, 4)
  )

  feat_tbl <- data.frame(
    id = 1:2,
    name = c("feature_1", "feature_2")
  )

  dist_feat_tbl <- data.frame(
    pu = c(1, 1, 2, 3, 4),
    feature = c(1, 2, 2, 1, 2),
    amount = c(5, 2, 3, 4, 1)
  )

  actions_df <- data.frame(
    id = "conservation",
    name = "conservation"
  )

  effects_df <- data.frame(
    pu = c(1, 2, 3, 4),
    action = "conservation",
```

```
  feature = c(1, 1, 2, 2),
  benefit = c(2, 1, 1, 2),
  loss = c(0, 0, 0, 0)
)

p <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  cost = "cost"
) |>
  add_actions(actions_df, cost = 0) |>
  add_effects(effects_df) |>
  add_constraint_targets_relative(0.2) |>
  add_objective_min_cost() |>
  set_solver_cbc(time_limit = 10)

sol <- solve(p)

get_actions(sol)
get_actions(sol, only_selected = TRUE)
}
```

`get_features` *Get feature summary from a solution*

Description

Extract the per-feature summary table from a `Solution` or `SolutionSet` object returned by `solve`.

The returned table summarizes, for each feature, the total amount available in the landscape together with the positive and negative contributions induced by the selected actions in the solution.

Usage

```
get_features(x, run = NULL)
```

Arguments

`x`	A `Solution` or `SolutionSet` object returned by `solve`.
`run`	Optional positive integer giving the run index to extract from a `SolutionSet`. If `NULL`, all runs are returned when available.

Details

This function reads the feature summary stored in `x$summary$features`. It errors if that table is missing.

Let B_f denote the total baseline amount available in the landscape for feature f. Let G_f denote the total positive contribution induced by selected actions, and let L_f denote the total negative contribution. Then the returned table is intended to summarize quantities of the form:

$$\text{net}_f = G_f - L_f,$$

$$\text{total}_f = B_f + \text{net}_f.$$

In the stored summary, these quantities are typically represented by the columns:

- `total_available`, corresponding to B_f,
- `benefit`, corresponding to G_f,
- `loss`, corresponding to L_f,
- `net`, corresponding to $G_f - L_f$,
- `total`, corresponding to $B_f + G_f - L_f$.

If any of the core numeric columns `total_available`, `benefit`, or `loss` are missing, they are created and filled with zero. If `net` is missing, it is computed as `benefit - loss`. If `total` is missing, it is computed as `total_available + net`.

Thus, this function guarantees that the returned table contains the columns `total_available`, `benefit`, `loss`, `net`, and `total`, even if some of them were absent from the stored summary.

If `x` is a `SolutionSet` and `run` is provided, only rows belonging to that run are returned. If the result contains a `run_id` column but only a single run is present and `run` was not requested explicitly, the `run_id` column is removed for convenience.

This function summarizes feature outcomes in the solution. It is different from `get_targets`, which focuses on target achievement rather than total feature balance.

Value

A `data.frame` with one row per feature, or one row per feature–run combination when multiple runs are present. The returned table always includes the columns `total_available`, `benefit`, `loss`, `net`, and `total`.

See Also

`get_pu`, `get_actions`, `get_targets`

Examples

```
if (requireNamespace("rcbc", quietly = TRUE)) {
  pu_tbl <- data.frame(
    id = 1:4,
    cost = c(1, 2, 3, 4)
  )

  feat_tbl <- data.frame(
    id = 1:2,
    name = c("feature_1", "feature_2")
  )

  dist_feat_tbl <- data.frame(
    pu = c(1, 1, 2, 3, 4),
    feature = c(1, 2, 2, 1, 2),
    amount = c(5, 2, 3, 4, 1)
  )

  actions_df <- data.frame(
    id = "conservation",
    name = "conservation"
  )

  effects_df <- data.frame(
    pu = c(1, 2, 3, 4),
    action = "conservation",
    feature = c(1, 1, 2, 2),
    benefit = c(2, 1, 1, 2),
    loss = c(0, 0, 0, 0)
  )

  p <- create_problem(
    pu = pu_tbl,
    features = feat_tbl,
    dist_features = dist_feat_tbl,
    cost = "cost"
  ) |>
    add_actions(actions_df, cost = 0) |>
    add_effects(effects_df) |>
    add_constraint_targets_relative(0.2) |>
    add_objective_min_cost() |>
    set_solver_cbc(time_limit = 10)

  sol <- solve(p)

  get_features(sol)
}
```

get_pu *Get planning-unit results from a solution*

Description

Extract the planning-unit summary table from a Solution or SolutionSet 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 the solution.

Usage

```
get_pu(x, only_selected = FALSE, run = NULL)
```

Arguments

x	A Solution or SolutionSet object returned by solve.
only_selected	Logical. If TRUE, return only rows where selected == 1. Default is FALSE.
run	Optional positive integer giving the run index to extract from a SolutionSet. If NULL, all runs are returned when available.

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 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 x is a SolutionSet and run is provided, only rows belonging to that run are returned. This requires the summary table to contain a run_id column.

If only_selected = TRUE, only rows with selected == 1 are returned. This requires the summary table to contain a selected column.

This function is intended for user-facing inspection of planning-unit results. For the raw model variable vector, use get_solution_vector.

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, get_solution_vector

Examples

```
if (requireNamespace("rcbc", quietly = TRUE)) {
  pu_tbl <- data.frame(
    id = 1:4,
    cost = c(1, 2, 3, 4)
  )

  feat_tbl <- data.frame(
    id = 1:2,
    name = c("feature_1", "feature_2")
  )

  dist_feat_tbl <- data.frame(
    pu = c(1, 1, 2, 3, 4),
    feature = c(1, 2, 2, 1, 2),
    amount = c(5, 2, 3, 4, 1)
  )

  actions_df <- data.frame(
    id = "conservation",
    name = "conservation"
  )

  effects_df <- data.frame(
    pu = c(1, 2, 3, 4),
    action = "conservation",
    feature = c(1, 1, 2, 2),
    benefit = c(2, 1, 1, 2),
    loss = c(0, 0, 0, 0)
  )

  p <- create_problem(
    pu = pu_tbl,
    features = feat_tbl,
    dist_features = dist_feat_tbl,
    cost = "cost"
  ) |>
    add_actions(actions_df, cost = 0) |>
    add_effects(effects_df) |>
    add_constraint_targets_relative(0.2) |>
    add_objective_min_cost() |>
    set_solver_cbc(time_limit = 10)

  sol <- solve(p)

  get_pu(sol)
  get_pu(sol, only_selected = TRUE)
}
```

`get_solution_vector` *Get raw solution vector from a solution*

Description

Return the raw decision-variable vector produced by the solver, in the internal model-variable order used by the optimization backend.

Usage

```
get_solution_vector(x, run = 1L)
```

Arguments

`x`	A `Solution` or `SolutionSet` object returned by `solve`.
`run`	Positive integer giving the run index to extract when `x` is a `SolutionSet`. Default is `1L`.

Details

This function extracts the raw solution vector stored at `x$solution$vector` for a `Solution` or for a selected run of a `SolutionSet`.

The returned vector is in the internal variable order of the optimization model. Depending on the problem formulation, it may include:

- planning-unit selection variables,
- action-allocation variables,
- auxiliary variables introduced for targets, budgets, fragmentation, or other constraints/objectives,
- and potentially additional blocks created internally by the model builder.

Therefore, this vector is primarily intended for advanced users, debugging, diagnostics, or internal verification. It is not a user-facing allocation table.

To inspect selected planning units or selected actions in a more interpretable form, use `get_pu` or `get_actions` instead.

For a single solution, the returned vector corresponds to that solution. For a `SolutionSet`, the `run` argument selects which run to extract.

Value

A numeric vector with one value per internal model variable.

See Also

`get_pu`, `get_actions`, `get_features`, `get_targets`

Examples

```
if (requireNamespace("rcbc", quietly = TRUE)) {
  pu_tbl <- data.frame(
    id = 1:4,
    cost = c(1, 2, 3, 4)
  )

  feat_tbl <- data.frame(
    id = 1:2,
    name = c("feature_1", "feature_2")
  )

  dist_feat_tbl <- data.frame(
    pu = c(1, 1, 2, 3, 4),
    feature = c(1, 2, 2, 1, 2),
    amount = c(5, 2, 3, 4, 1)
  )

  actions_df <- data.frame(
    id = "conservation",
    name = "conservation"
  )

  effects_df <- data.frame(
    pu = c(1, 2, 3, 4),
    action = "conservation",
    feature = c(1, 1, 2, 2),
    benefit = c(2, 1, 1, 2),
    loss = c(0, 0, 0, 0)
  )

  p <- create_problem(
    pu = pu_tbl,
    features = feat_tbl,
    dist_features = dist_feat_tbl,
    cost = "cost"
  ) |>
    add_actions(actions_df, cost = 0) |>
    add_effects(effects_df) |>
    add_constraint_targets_relative(0.2) |>
    add_objective_min_cost() |>
    set_solver_cbc(time_limit = 10)

  sol <- solve(p)

  v <- get_solution_vector(sol)
  v
  length(v)
}
```

get_targets *Get target achievement summary from a solution*

Description

Extract a user-facing target-achievement table from a Solution or SolutionSet object returned by solve.

The returned table summarizes, for each stored target, the target level, the achieved value in the solution, the gap between achieved and required values, and whether the target was met.

Usage

```
get_targets(x, run = NULL)
```

Arguments

x	A Solution or SolutionSet object returned by solve.
run	Optional positive integer giving the run index to extract from a SolutionSet. If NULL, all runs are returned when available.

Details

Targets are optional in multiscape. If the solution 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 x is a SolutionSet and run is provided, only rows belonging to that run are returned. If the result contains a run_id column but only a single run is present and run was not requested explicitly, the run_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 solution does not contain targets. Typical columns include `feature`, `feature_name`, `target_level`, `total_available`, `target`, `achieved`, `gap`, and `met`.

See Also

`get_pu`, `get_actions`, `get_features`

Examples

```
if (requireNamespace("rcbc", quietly = TRUE)) {
  pu_tbl <- data.frame(
    id = 1:4,
    cost = c(1, 2, 3, 4)
  )

  feat_tbl <- data.frame(
    id = 1:2,
    name = c("feature_1", "feature_2")
  )

  dist_feat_tbl <- data.frame(
    pu = c(1, 1, 2, 3, 4),
    feature = c(1, 2, 2, 1, 2),
    amount = c(5, 2, 3, 4, 1)
  )

  actions_df <- data.frame(
    id = "conservation",
    name = "conservation"
  )

  effects_df <- data.frame(
    pu = c(1, 2, 3, 4),
    action = "conservation",
    feature = c(1, 1, 2, 2),
    benefit = c(2, 1, 1, 2),
    loss = c(0, 0, 0, 0)
  )

  p <- create_problem(
    pu = pu_tbl,
    features = feat_tbl,
    dist_features = dist_feat_tbl,
    cost = "cost"
  ) |>
    add_actions(actions_df, cost = 0) |>
    add_effects(effects_df) |>
    add_constraint_targets_relative(0.2) |>
    add_objective_min_cost() |>
    set_solver_cbc(time_limit = 10)
```

```
  sol <- solve(p)

  get_targets(sol)
}
```

`load_sim_features_raster` *Example feature raster*

Description

Load the example feature raster shipped with the package.

Usage

```
load_sim_features_raster()
```

Value

A `terra::SpatRaster`.

`plot_spatial` *Plot spatial outputs from a solution or solution set*

Description

Convenience wrapper to plot spatial outputs from a `Solution` or `SolutionSet`.

Depending on `what`, this function dispatches to one of:

- `plot_spatial_pu`,
- `plot_spatial_actions`,
- `plot_spatial_features`.

This wrapper is useful as a compact entry point, while the specialised plotting functions provide a cleaner and more explicit user interface for each spatial output type.

Usage

```
plot_spatial(
  x,
  what = c("pu", "actions", "features"),
  runs = NULL,
  actions = 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_values = NULL,
  fill_na = "grey80",
  use_viridis = TRUE
)
```

Arguments

x	A Solution or SolutionSet object.
what	Character string indicating what to plot. Must be one of "pu", "actions", or "features".
runs	Optional integer vector of run ids. If NULL, a Solution is used directly and a SolutionSet defaults to the first run.
actions	Optional action subset used when what = "actions".
features	Optional feature subset used when what = "features".
value	Character string used only when what = "features". 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 "single" for planning units and actions, and "facet" for features.
max_facets	Maximum number of facets shown when faceting without an explicit action or feature subset.
...	Additional arguments passed to the specialised plotting function.
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 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 maps.
fill_na	Fill colour for missing values.
use_viridis	Logical. If TRUE and the **viridis** package is available, use viridis scales.

Value

Invisibly returns a ggplot object.

See Also

plot_spatial_pu, plot_spatial_actions, plot_spatial_features

Examples

```
if (requireNamespace("sf", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  data("sim_pu_sf", package = "multiscape")

  problem <- structure(
    list(
      data = list(
        pu_sf = sim_pu_sf
      )
    ),
    class = "Problem"
  )

  sol <- structure(
    list(
      problem = problem,
      summary = list(
        pu = data.frame(
          id = sim_pu_sf$id,
          selected = as.integer(seq_len(nrow(sim_pu_sf)) %% 2 == 1)
        )
      )
    ),
    class = "Solution"
  )

  plot_spatial(sol, what = "pu")
}
```

`plot_spatial_actions` *Plot selected actions in space*

Description

Plot the spatial distribution of selected actions from a `Solution` or `SolutionSet`.

This function maps the selected planning unit–action pairs returned by `get_actions(only_selected = TRUE)` onto the planning-unit geometry stored in the associated `Problem` object.

Usage

```
plot_spatial_actions(
  x,
  runs = NULL,
  actions = NULL,
  layout = NULL,
  max_facets = 4L,
  ...,
  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 `Solution` or `SolutionSet` object.
`runs`	Optional integer vector of run ids. If `NULL`, a `Solution` is used directly and a `SolutionSet` defaults to the first run.
`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.
`...`	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 x$problem$data$pu_sf.

Value

Invisibly returns a ggplot object.

See Also

get_actions, plot_spatial, plot_spatial_pu, plot_spatial_features

Examples

```
if (requireNamespace("sf", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  data("sim_pu_sf", package = "multiscape")

  actions_df <- data.frame(
    id = c("conservation", "restoration"),
    name = c("conservation", "restoration")
  )

  problem <- structure(
    list(
      data = list(
        pu_sf = sim_pu_sf,
        actions = actions_df
      )
    ),
    class = "Problem"
```

```
  )

  ids <- sim_pu_sf$id[seq_len(min(6, nrow(sim_pu_sf)))]
  sol <- structure(
    list(
      problem = problem,
      summary = list(
        actions = data.frame(
          pu = c(ids[1:3], ids[4:6]),
          action = c(rep("conservation", 3), rep("restoration", 3)),
          selected = 1L
        )
      )
    ),
    class = "Solution"
  )

  plot_spatial_actions(sol, layout = "facet")
}
```

`plot_spatial_features` *Plot spatial feature values from a solution*

Description

Plot feature values in space from a `Solution` or `SolutionSet`.

This function combines baseline feature amounts from `x$problem$data$dist_features` with positive effects induced by selected actions to produce planning-unit-level feature maps.

Usage

```
plot_spatial_features(
  x,
  runs = 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 Solution or SolutionSet object.
runs	Optional integer vector of run ids. If NULL, a Solution is used directly and a SolutionSet defaults to the first run.
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 x$problem$data$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 x$problem$data$pu_sf.

Value

Invisibly returns a ggplot object.

See Also

get_features, plot_spatial, plot_spatial_pu, plot_spatial_actions

Examples

```
if (requireNamespace("sf", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  data("sim_pu_sf", package = "multiscape")

  n <- min(6, nrow(sim_pu_sf))
  ids <- sim_pu_sf$id[seq_len(n)]

  features_df <- data.frame(
    id = c(1, 2),
    name = c("feature_1", "feature_2")
  )

  dist_features_df <- data.frame(
    pu = rep(ids, times = 2),
    feature = rep(c(1, 2), each = n),
    amount = c(seq_len(n), rev(seq_len(n)))
  )

  dist_effects_df <- data.frame(
    pu = ids,
    action = "conservation",
    feature = rep(c(1, 2), length.out = n),
    benefit = rep(1, n)
  )

  actions_df <- data.frame(
    id = "conservation",
    name = "conservation"
  )

  problem <- structure(
    list(
      data = list(
        pu_sf = sim_pu_sf,
        features = features_df,
        dist_features = dist_features_df,
        dist_effects = dist_effects_df,
        actions = actions_df
      )
    ),
    class = "Problem"
  )
```

```
  sol <- structure(
    list(
      problem = problem,
      summary = list(
        actions = data.frame(
          pu = ids,
          action = "conservation",
          selected = 1L
        )
      )
    ),
    class = "Solution"
  )

  plot_spatial_features(sol, features = "feature_1", value = "final")
}
```

`plot_spatial_pu` *Plot selected planning units in space*

Description

Plot the spatial distribution of selected planning units from a `Solution` or `SolutionSet`.

This function maps the planning-unit selection summary returned by `get_pu` onto the planning-unit geometry stored in the associated `Problem` object.

Usage

```
plot_spatial_pu(
  x,
  runs = 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 `Solution` or `SolutionSet` object.
`runs`	Optional integer vector of run ids. If `NULL`, a `Solution` is used directly and a `SolutionSet` defaults to the first run.
`...`	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 x$problem$data$pu_sf.

Value

Invisibly returns a ggplot object.

See Also

get_pu, plot_spatial, plot_spatial_actions, plot_spatial_features

Examples

```
if (requireNamespace("sf", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  data("sim_pu_sf", package = "multiscape")

  problem <- structure(
    list(
      data = list(
        pu_sf = sim_pu_sf
      )
    ),
    class = "Problem"
  )

  sol <- structure(
    list(
      problem = problem,
      summary = list(
        pu = data.frame(
          id = sim_pu_sf$id,
          selected = as.integer(seq_len(nrow(sim_pu_sf)) %% 2 == 1)
        )
      )
    ),
```

```
    class = "Solution"
  )

  plot_spatial_pu(sol)
}
```

`plot_tradeoff` *Plot trade-offs from a multi-objective solution set*

Description

Plot pairwise trade-offs among objective values stored in a `SolutionSet`.

This function is intended for multi-objective workflows in which the solution set contains one row per run and one 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` 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, get_solution_vector

Examples

```
if (requireNamespace("ggplot2", quietly = TRUE)) {
  solset <- structure(
    list(
      solution = list(
        runs = data.frame(
          run_id = 1:5,
          status = rep("optimal", 5),
          runtime = c(1.2, 1.1, 1.4, 1.3, 1.5),
          gap = c(0, 0, 0, 0, 0),
          value_cost = c(10, 12, 14, 16, 18),
          value_benefit = c(5, 7, 8, 9, 10),
          value_loss = c(4, 3, 3, 2, 2)
        )
      )
    ),
    class = "SolutionSet"
  )

  # Plot all pairwise trade-offs
  plot_tradeoff(solset)

  # Plot two selected objectives
  plot_tradeoff(solset, objectives = c("cost", "benefit"))

  # Colour points by one objective and label runs
  plot_tradeoff(solset, color_by = "loss", label_runs = TRUE)
}
```

---

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

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.

`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.

`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_effects`, `add_constraint_targets_absolute`, `solve`

`set_method_augmecon` *Set the AUGMECON multi-objective method*

Description

Configure a `Problem` object to be solved with the augmented epsilon-constraint method (AUGMECON).

AUGMECON is an exact multi-objective optimization method in which one objective is treated as the primary objective and the remaining objectives are converted into ε-constraints. In the augmented formulation, each secondary objective is associated with a non-negative slack variable, and the primary objective is augmented with a small reward term based on the normalized slacks. This augmentation is used to avoid weakly efficient solutions, following Mavrotas (2009).

This function does not solve the problem directly. It stores the AUGMECON configuration in `x$data$method`, to be used later by `solve`.

Usage

```
set_method_augmecon(
  x,
  primary,
  aliases = NULL,
  grid = NULL,
  n_points = 10,
  include_extremes = TRUE,
  lexicographic = TRUE,
  lexicographic_tol = 1e-09,
  augmentation = 0.001,
  slack_upper_bound = 1e+06
)
```

Arguments

`x`	A `Problem` object.
`primary`	Character string giving the alias of the primary objective, that is, the objective optimized directly in the AUGMECON formulation.
`aliases`	Optional character vector of objective aliases to include in the method. If `NULL`, all registered objective aliases are used. The value of `primary` must be included in `aliases`.
`grid`	Optional named list defining manual epsilon levels for the secondary objectives. Each name must correspond to a secondary objective alias, and each element must be a non-empty numeric vector of finite values. If `NULL`, the grid is generated automatically.
`n_points`	Number of automatically generated epsilon levels per secondary objective when `grid = NULL`. Must be at least 2. Ignored when `grid` is supplied.
`include_extremes`	Logical. If `TRUE`, automatically generated grids include extreme values of each secondary objective.
`lexicographic`	Logical. If `TRUE`, use lexicographic anchoring when computing extreme points for automatic grid construction.
`lexicographic_tol`	Non-negative numeric tolerance used in lexicographic anchoring.
`augmentation`	Positive numeric augmentation coefficient ρ. The effective coefficient of each secondary slack is computed internally as ρ / R_k, where R_k is the payoff-table range of the corresponding secondary objective.
`slack_upper_bound`	Positive numeric upper bound imposed on the explicit non-negative slack variables associated with the secondary objectives.

Details

Use this method when one objective should be optimized directly, the remaining objectives should be controlled through epsilon levels, and weakly efficient solutions should be reduced through the augmented formulation.

General idea

Suppose that $m \geq 2$ objective functions have already been registered in the problem:

$$f_1(x), f_2(x), \ldots, f_m(x).$$

AUGMECON selects one of them as the primary objective, say $f_p(x)$, and treats the remaining $m-1$ objectives as secondary objectives.

For a fixed combination of epsilon levels, the method solves a sequence of single-objective subproblems of the form:

$$\max \; f_p(x) + \rho \sum_{k \in \mathcal{S}} \frac{s_k}{R_k}$$

subject to

$$f_k(x) - s_k = \varepsilon_k, \qquad k \in \mathcal{S},$$

$$s_k \geq 0, \qquad k \in \mathcal{S},$$

together with all original feasibility constraints of the planning problem.

Here:

- $f_p(x)$ is the primary objective,
- $\mathcal{S}$ is the set of secondary objectives,
- ε_k is the imposed level for secondary objective k,
- s_k is a non-negative slack variable,
- R_k is the payoff-table range used to normalize objective k,
- $\rho > 0$ is a small augmentation coefficient.

In the original AUGMECON formulation of Mavrotas (2009), the augmentation term ensures that, among solutions with the same primary objective value, the solver prefers those with larger normalized slack, thereby avoiding weakly efficient points and improving Pareto-front generation.

Secondary-objective equalities and slacks

The key difference between standard epsilon-constraint and AUGMECON is that the secondary objectives are written as equalities with slacks rather than as simple inequalities. For a maximization-type secondary objective, this takes the form:

$$f_k(x) - s_k = \varepsilon_k, \qquad s_k \geq 0.$$

This implies:

$$f_k(x) \geq \varepsilon_k,$$

while explicitly measuring the excess above the imposed epsilon level through s_k. The augmentation term then rewards such excess in normalized form.

In implementation terms, the exact sign convention for each objective depends on whether it is internally treated as a minimization or maximization objective, but the method always preserves the same AUGMECON principle:

- one objective is optimized directly,
- all others are turned into constrained objectives,
- non-negative slacks measure controlled deviation from the imposed epsilon levels,
- the primary objective is augmented with a small slack-based reward.

Manual and automatic epsilon grids

AUGMECON requires a grid of epsilon levels for each secondary objective.

If `grid` is supplied, it must be a named list with one numeric vector per secondary objective. Each vector defines the exact epsilon levels to be explored for that objective.

If `grid = NULL`, the grid is generated automatically later during `solve`. In that case, the method first computes extreme points and payoff-table ranges for the secondary objectives, and then generates `n_points` levels for each one.

If `include_extremes = TRUE`, the automatic grid includes the extreme values of each secondary objective.

If `lexicographic = TRUE`, extreme points are computed using lexicographic anchoring, which can improve payoff-table quality when objectives are tightly competing. The tolerance used for lexicographic anchoring is controlled by `lexicographic_tol`.

Normalization and augmentation

The augmentation term is scaled using the payoff-table ranges of the secondary objectives. If R_k denotes the range of secondary objective k, then the effective coefficient applied to the slack is:

$$\frac{\rho}{R_k},$$

where $\rho =$ `augmentation`.

This normalization is important because different objectives may be measured on very different numerical scales. Without normalization, a slack belonging to a large-scale objective could dominate the augmentation term simply due to units.

In this implementation, the user supplies `augmentation` as the base coefficient ρ, while the normalized slack coefficients are computed internally at solve time using the corresponding payoff-table ranges.

Stored configuration

This function stores the method definition in `x$data$method` with:

- `name = "augmecon"`,
- the primary objective alias,
- the full set of participating aliases,
- the set of secondary aliases,
- either a manual grid or the information required to generate one automatically,
- augmentation and slack-bound parameters.

The actual payoff table, grid construction, and subproblem solution loop are performed later by `solve`.

Value

The updated `Problem` object with the AUGMECON method configuration stored in `x$data$method`.

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.

See Also

`set_method_epsilon_constraint`, `set_method_weighted_sum`, `solve`

Examples

```
# Small toy problem
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)

feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)

dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)

actions_df <- data.frame(
  id = c("conservation", "restoration"),
  name = c("conservation", "restoration")
)

effects_df <- data.frame(
  pu = c(1, 2, 3, 4, 1, 2, 3, 4),
  action = c("conservation", "conservation", "conservation", "conservation",
             "restoration", "restoration", "restoration", "restoration"),
  feature = c(1, 1, 1, 1, 2, 2, 2, 2),
  benefit = c(2, 1, 0, 1, 3, 0, 1, 2),
  loss = c(0, 0, 1, 0, 0, 1, 0, 0)
)

x <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  cost = "cost"
) |>
  add_actions(actions_df, cost = c(conservation = 1, restoration = 2)) |>
```

```
  add_effects(effects_df) |>
  add_objective_max_benefit(alias = "benefit") |>
  add_objective_min_cost(alias = "cost") |>
  add_objective_min_loss(alias = "loss")

# Automatic epsilon grids generated later during solve()
x1 <- set_method_augmecon(
  x,
  primary = "benefit",
  aliases = c("benefit", "cost"),
  n_points = 5,
  include_extremes = TRUE,
  lexicographic = TRUE,
  augmentation = 1e-3
)

x1$data$method

# Manual epsilon grids for two secondary objectives
x2 <- set_method_augmecon(
  x,
  primary = "benefit",
  aliases = c("benefit", "cost", "loss"),
  grid = list(
    cost = c(4, 6, 8),
    loss = c(0, 1)
  ),
  augmentation = 1e-3,
  slack_upper_bound = 1e6
)

x2$data$method
```

`set_method_epsilon_constraint` *Set the epsilon-constraint multi-objective method*

Description

Configure a `Problem` object to be solved with the epsilon-constraint multi-objective method.

In this method, one objective is designated as the *primary* objective and is optimized directly, while the remaining objectives are transformed into ε-constraints.

Two operating modes are supported:

- **manual mode**: the user supplies the ε-levels explicitly,
- **automatic mode**: the ε-levels are generated later during `solve` from extreme-point or payoff information.

This function does not solve the problem. It stores the method configuration in `x$data$method`, to be used later by `solve`.

Usage

```
set_method_epsilon_constraint(
  x,
  primary,
  eps = NULL,
  aliases = NULL,
  mode = c("manual", "auto"),
  n_points = 10,
  include_extremes = TRUE,
  lexicographic = TRUE,
  lexicographic_tol = 1e-08
)
```

Arguments

x	A Problem object.
primary	Character string giving the alias of the primary objective to optimize directly.
eps	Optional epsilon specification used only in mode = "manual". It may be: • a named numeric vector, defining epsilon values for a single run, • or a named list of numeric vectors, defining epsilon values for a grid of runs. Names must correspond exactly to the constrained objective aliases.
aliases	Optional character vector of objective aliases to include. By default, all registered objective aliases are used. The value of primary must be included in aliases.
mode	Character string. Must be either "manual" or "auto".
n_points	Integer scalar used only in mode = "auto". Number of epsilon points to generate automatically for the constrained objective. Must be at least 2.
include_extremes	Logical scalar used only in mode = "auto". If TRUE, include extreme epsilon values in the automatically generated grid.
lexicographic	Logical scalar used only in mode = "auto". If TRUE, compute extreme points lexicographically.
lexicographic_tol	Numeric scalar ≥ 0. Tolerance used in lexicographic extreme-point computation.

Details

Use this method when one objective should be optimized directly while the remaining objectives are controlled through explicit performance thresholds.

General idea

Suppose that $m \geq 2$ objective functions have already been registered in the problem:

$$f_1(x), f_2(x), \ldots, f_m(x).$$

The epsilon-constraint method selects one of them as the primary objective, say $f_p(x)$, and treats the remaining objectives as constrained objectives.

For a fixed vector of epsilon levels, the method solves subproblems in which the primary objective is optimized directly and the remaining objectives are imposed through epsilon constraints.

A representative formulation is:

$$\max \; f_p(x)$$

subject to

$$f_k(x) \geq \varepsilon_k, \qquad k \in \mathcal{C},$$

together with all original feasibility constraints of the planning problem, where $\mathcal{C}$ is the set of constrained objectives.

Depending on objective sense, the internal implementation may transform minimization objectives into equivalent constrained forms, but the method always follows the same principle:

- one objective is optimized directly,
- all remaining objectives are imposed through ε-constraints.

By solving the problem repeatedly for different epsilon levels, the method generates a set of efficient trade-off solutions.

Atomic objectives requirement

The epsilon-constraint method can only be used with atomic objectives that have already been registered under aliases. These aliases are typically created by calling objective setters with an `alias` argument, for example:

```
x <- x |>
  add_objective_max_benefit(alias = "benefit") |>
  add_objective_min_cost(alias = "cost") |>
  add_objective_min_fragmentation(alias = "frag")
```

The `primary` argument selects which registered objective is optimized directly. The remaining aliases are treated as constrained objectives.

Manual mode

In `mode = "manual"`, the user must provide `eps`.

The `eps` argument can be supplied as:

- a named numeric vector, defining a single run,
- or a named list of numeric vectors, defining a grid of runs.

The names of `eps` must correspond exactly to the constrained objective aliases, that is, to all aliases in `aliases` except `primary`.

If the constrained objectives are $\mathcal{C} = \{c_1, \ldots, c_q\}$, then manual mode creates a design grid containing all combinations of the supplied epsilon levels for the constrained objectives.

Each row of this grid defines one subproblem to be solved later.

Important: manual mode supports **two or more objectives**. In particular, it can be used with:

- 2 objectives: 1 primary + 1 constrained objective,
- 3 or more objectives: 1 primary + multiple constrained objectives.

Thus, manual mode is the general way to use the epsilon-constraint method when more than two objectives are involved.

In manual mode, the generated design grid is stored immediately in `x$data$method$runs`. Its epsilon columns are named `eps_<alias>`, for example `eps_frag`.

Automatic mode

In `mode = "auto"`, the user omits `eps` and instead supplies `n_points`.

In this case, the epsilon grid is not built immediately. Instead, it is constructed later during `solve` using extreme-point or payoff-table information.

In the current implementation, automatic mode supports **exactly two objectives only**:

- one primary objective,
- one constrained objective.

Therefore, if `mode = "auto"`, then `aliases` must contain exactly two objective aliases. Problems with three or more objectives must use `mode = "manual"`.

If `include_extremes = TRUE`, the automatically generated grid includes the extreme values of the constrained objective. Otherwise, only interior values are used.

If `lexicographic = TRUE`, the extreme points used to generate the grid are computed lexicographically. In that case, one objective is optimized first, and then the second objective is optimized while constraining the first to remain within `lexicographic_tol` of its optimum.

Stored configuration

The configured method stores:

- `name = "epsilon_constraint"`,
- `mode`,
- `primary`,
- `aliases`,
- `constrained`,
- epsilon design information,
- lexicographic configuration.

In manual mode, `x$data$method$runs` contains the explicit design grid.

In automatic mode, `x$data$method$runs` is initially `NULL` and is generated later during `solve`.

For more than two objectives, automatic grid generation is currently unavailable because the number of epsilon combinations grows rapidly and requires explicit user control.

Value

An updated `Problem` object with the epsilon-constraint method configuration stored in `x$data$method`.

In manual mode, `x$data$method$runs` contains the explicit epsilon-design grid.

In automatic mode, `x$data$method$runs` is `NULL` until the grid is generated later during `solve`.

See Also

set_method_augmecon, set_method_weighted_sum, solve

Examples

```
# Small toy problem
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)

feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)

dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)

actions_df <- data.frame(
  id = c("conservation", "restoration"),
  name = c("conservation", "restoration")
)

effects_df <- data.frame(
  pu = c(1, 2, 3, 4, 1, 2, 3, 4),
  action = c("conservation", "conservation", "conservation", "conservation",
             "restoration", "restoration", "restoration", "restoration"),
  feature = c(1, 1, 1, 1, 2, 2, 2, 2),
  benefit = c(2, 1, 0, 1, 3, 0, 1, 2),
  loss = c(0, 0, 1, 0, 0, 1, 0, 0)
)

x <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  cost = "cost"
) |>
  add_actions(actions_df, cost = c(conservation = 1, restoration = 2)) |>
  add_effects(effects_df) |>
  add_objective_min_cost(alias = "cost") |>
  add_objective_max_benefit(alias = "benefit") |>
  add_objective_min_loss(alias = "loss")

# Manual mode with one constrained objective
x1 <- set_method_epsilon_constraint(
  x,
  primary = "benefit",
```

```
  aliases = c("benefit", "cost"),
  mode = "manual",
  eps = c(cost = 5)
)

x1$data$method

# Manual mode with multiple epsilon values
x2 <- set_method_epsilon_constraint(
  x,
  primary = "benefit",
  aliases = c("benefit", "cost"),
  mode = "manual",
  eps = list(cost = c(4, 6, 8))
)

x2$data$method$runs

# Manual mode with more than two objectives
x3 <- set_method_epsilon_constraint(
  x,
  primary = "benefit",
  aliases = c("benefit", "cost", "loss"),
  mode = "manual",
  eps = list(
    cost = c(4, 6),
    loss = c(0, 1)
  )
)

x3$data$method$runs

# Automatic mode currently supports exactly two objectives
x4 <- set_method_epsilon_constraint(
  x,
  primary = "benefit",
  aliases = c("benefit", "cost"),
  mode = "auto",
  n_points = 5,
  include_extremes = TRUE,
  lexicographic = TRUE,
  lexicographic_tol = 1e-8
)

x4$data$method
```

---

set_method_weighted_sum

Set the weighted-sum multi-objective method

Description

Configure a `Problem` object to be solved with a weighted-sum multi-objective method.

In the weighted-sum method, several registered atomic objectives are combined into a single scalar objective using a weighted linear combination. This function stores that configuration in `x$data$method` so that it can be used later by `solve`.

Usage

```
set_method_weighted_sum(
  x,
  aliases,
  weights,
  normalize_weights = FALSE,
  objective_scaling = FALSE
)
```

Arguments

`x`	A `Problem` object.
`aliases`	Character vector of objective aliases to combine. Each alias must correspond to a previously registered atomic objective.
`weights`	Numeric vector of weights, with the same length and order as `aliases`. Weights must be finite. If `normalize_weights = TRUE`, they are rescaled to sum to 1 before being stored.
`normalize_weights`	Logical. If `TRUE`, normalize the supplied weights to sum to 1 before storing them.
`objective_scaling`	Logical. If `TRUE`, request scaling of the participating objectives before weighted aggregation in the solving layer.

Details

Use this method when several registered objectives should be combined into a single scalar optimization problem through explicit preference weights.

General idea

Suppose that a set of atomic objectives has already been registered in the problem under aliases $k \in \mathcal{K}$. Let $f_k(x)$ denote the scalar value of objective k, and let w_k denote its user-supplied weight.

The weighted-sum method combines them into a single scalar objective of the form:

$$\sum_{k \in \mathcal{K}} w_k \, f_k(x).$$

In practice, the exact sign convention used internally depends on the sense of each registered atomic objective, for example whether it is a minimization-type or maximization-type objective. The solving layer is therefore responsible for constructing a solver-ready scalar objective from the stored objective specifications and the requested weights.

Atomic objectives requirement

The weighted-sum method can only be used with atomic objectives that have already been registered under aliases. These aliases are typically created by calling objective setters with an `alias` argument, for example:

```
x <- x |>
  add_objective_min_cost(alias = "cost") |>
  add_objective_min_fragmentation(alias = "frag")
```

Internally, each atomic objective is stored in `x$data$objectives[[alias]]` together with its metadata, such as:

- `objective_id`,
- `model_type`,
- `sense`,
- `objective_args`.

The `aliases` argument passed to this function selects which of those registered atomic objectives are included in the weighted combination.

Weight normalization

If `normalize_weights = TRUE`, the supplied weights are rescaled to sum to 1:

$$\tilde{w}_k = \frac{w_k}{\sum_{j \in \mathcal{K}} w_j}.$$

This normalization does not change the optimizer's solution in a pure weighted-sum formulation as long as all weights are multiplied by the same positive constant, but it can improve interpretability and numerical conditioning.

If `normalize_weights = FALSE`, the supplied weights are stored exactly as provided.

Objective scaling

If `objective_scaling = TRUE`, the solving layer is expected to scale the participating objectives before combining them. The purpose of scaling is to reduce distortions caused by objectives being measured on very different numerical ranges.

Conceptually, if R_k denotes a scale or range associated with objective k, then a scaled weighted sum may be interpreted as:

$$\sum_{k \in \mathcal{K}} w_k \, \frac{f_k(x)}{R_k}.$$

The exact scaling rule is implemented later in the solving layer. This function only stores whether objective scaling has been requested.

Mixed objective senses

Weighted sums are straightforward when all participating objectives have the same optimization sense. When minimization and maximization objectives are mixed, the solving layer must standardize them internally before building the scalar objective.

This function validates that the requested aliases exist, but it does not itself resolve objective-sense compatibility. That logic is delegated to the downstream solving layer.

Theoretical limitation

The weighted-sum method typically recovers only *supported* efficient solutions, that is, solutions lying on the convex hull of the Pareto front in objective space. In non-convex multi-objective problems, especially mixed integer problems, some efficient solutions cannot be obtained by any weighted combination. In such cases, methods such as `set_method_epsilon_constraint` or `set_method_augmecon` may be preferable.

Stored configuration

This function stores the method definition in `x$data$method` with:

- `name = "weighted"`,
- `aliases`,
- `weights`,
- `normalize_weights`,
- `objective_scaling`.

The actual scalarization is performed later by `solve`.

Value

The updated `Problem` object with the weighted-sum method configuration stored in `x$data$method`.

See Also

`set_method_epsilon_constraint`, `set_method_augmecon`, `solve`

Examples

```
# Small toy problem
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)

feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)

dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)

actions_df <- data.frame(
  id = c("conservation", "restoration"),
  name = c("conservation", "restoration")
```

```
)

effects_df <- data.frame(
  pu = c(1, 2, 3, 4, 1, 2, 3, 4),
  action = c("conservation", "conservation", "conservation", "conservation",
             "restoration", "restoration", "restoration", "restoration"),
  feature = c(1, 1, 1, 1, 2, 2, 2, 2),
  benefit = c(2, 1, 0, 1, 3, 0, 1, 2),
  loss = c(0, 0, 1, 0, 0, 1, 0, 0)
)

x <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  cost = "cost"
) |>
  add_actions(actions_df, cost = c(conservation = 1, restoration = 2)) |>
  add_effects(effects_df) |>
  add_objective_min_cost(alias = "cost") |>
  add_objective_max_benefit(alias = "benefit")

x <- set_method_weighted_sum(
  x,
  aliases = c("cost", "benefit"),
  weights = c(0.4, 0.6),
  normalize_weights = FALSE
)

x$data$method

# Normalize weights before storing
x2 <- set_method_weighted_sum(
  x,
  aliases = c("cost", "benefit"),
  weights = c(2, 3),
  normalize_weights = TRUE
)

x2$data$method

# Request objective scaling
x3 <- set_method_weighted_sum(
  x,
  aliases = c("cost", "benefit"),
  weights = c(0.7, 0.3),
  objective_scaling = TRUE
)

x3$data$method
```

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 = c("auto", "gurobi", "cplex", "cbc", "symphony"),
  gap_limit = NULL,
  time_limit = NULL,
  solution_limit = NULL,
  cores = NULL,
  verbose = FALSE,
  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 controlling backend-specific early stopping after feasible solution discovery. If NULL, the previously stored value is kept unchanged.
cores	Optional positive integer giving the number of CPU cores to use. 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 name of the solver log file. If NULL, the previously stored value is kept unchanged.
write_log	Optional logical flag indicating whether solver output should be written to a file. If NULL, the previously stored value is kept unchanged.

solver_params	Named list of solver-specific parameters. These are merged with previously stored backend-specific parameters rather than replacing them completely.
...	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.

Incremental update semantics

This function updates solver settings incrementally.

If an argument is supplied as NULL, the previously stored value is kept unchanged. Therefore, repeated calls can be used to modify only selected components of the solver configuration.

For example, a user may first configure the solver backend and time limit, and later update only the optimality gap or only a backend-specific parameter.

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 number of CPU cores to use. If the requested number exceeds the number of detected cores, 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-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, and the result is then merged with any previously stored solver_params. Existing parameters are therefore preserved unless explicitly overwritten.

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

```
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)

feat_tbl <- data.frame(
```

```
  id = 1:2,
  name = c("feature_1", "feature_2")
)

dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)

x <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  cost = "cost"
)

x1 <- set_solver(
  x,
  solver = "cbc",
  gap_limit = 0.01,
  time_limit = 300,
  cores = 2,
  verbose = TRUE
)

x1$data$solve_args

# Update only selected settings
x2 <- set_solver(
  x1,
  gap_limit = 0.05,
  solver_params = list(randomSeed = 123)
)

x2$data$solve_args
```

`set_solver_cbc` *Configure CPLEX solver settings*

Description

Convenience wrapper around `set_solver` that stores `solver = "cplex"` in the problem object.

This function does not solve the model. It only updates the stored solver configuration.

Usage

```
set_solver_cbc(
  x,
```

```
  ...,
  solver_params = list(),
  gap_limit = NULL,
  time_limit = NULL,
  solution_limit = NULL,
  cores = NULL,
  verbose = FALSE,
  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 backend-specific parameters rather than replacing them completely.
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 controlling backend-specific early stopping after feasible solution discovery. If NULL, the previously stored value is kept unchanged.
cores	Optional positive integer giving the number of CPU cores to use. 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 name of the solver log file. If NULL, the previously stored value is kept unchanged.
write_log	Optional logical flag indicating whether solver output should be written to a file. If NULL, the previously stored value is kept unchanged.

Value

An updated Problem object with CPLEX solver settings stored in x$data$solve_args.

See Also

set_solver, solve

Examples

```
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)
```

```
feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)

dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)

x <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  cost = "cost"
)

x <- set_solver_cbc(
  x,
  gap_limit = 0.01,
  time_limit = 300,
  cores = 2
)

x$data$solve_args
```

`set_solver_cplex` *Configure CPLEX solver settings*

Description

Convenience wrapper around `set_solver` that sets `solver = "cplex"`.

Usage

```
set_solver_cplex(
  x,
  ...,
  solver_params = list(),
  gap_limit = NULL,
  time_limit = NULL,
  solution_limit = NULL,
  cores = NULL,
  verbose = FALSE,
  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 backend-specific parameters rather than replacing them completely.
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 controlling backend-specific early stopping after feasible solution discovery. If NULL, the previously stored value is kept unchanged.
cores	Optional positive integer giving the number of CPU cores to use. 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 name of the solver log file. If NULL, the previously stored value is kept unchanged.
write_log	Optional logical flag indicating whether solver output should be written to a file. If NULL, the previously stored value is kept unchanged.

Value

An updated Problem object with CPLEX solver settings.

See Also

set_solver, solve

Examples

```
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)

feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)

dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)
```

```
x <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  cost = "cost"
)

x <- set_solver_cplex(
  x,
  gap_limit = 0.001,
  time_limit = 1200,
  cores = 2
)

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.

Usage

```
set_solver_gurobi(
  x,
  ...,
  solver_params = list(),
  gap_limit = NULL,
  time_limit = NULL,
  solution_limit = NULL,
  cores = NULL,
  verbose = FALSE,
  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 backend-specific parameters rather than replacing them completely.

`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 controlling backend-specific early stopping after feasible solution discovery. If `NULL`, the previously stored value is kept unchanged.
`cores`	Optional positive integer giving the number of CPU cores to use. 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 name of the solver log file. If `NULL`, the previously stored value is kept unchanged.
`write_log`	Optional logical flag indicating whether solver output should be written to a file. 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

```
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)

feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)

dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)

x <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  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.

Usage

```
set_solver_symphony(
  x,
  ...,
  solver_params = list(),
  gap_limit = NULL,
  time_limit = NULL,
  solution_limit = NULL,
  cores = NULL,
  verbose = FALSE,
  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 backend-specific parameters rather than replacing them completely.
`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 controlling backend-specific early stopping after feasible solution discovery. If `NULL`, the previously stored value is kept unchanged.

cores	Optional positive integer giving the number of CPU cores to use. 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 name of the solver log file. If NULL, the previously stored value is kept unchanged.
write_log	Optional logical flag indicating whether solver output should be written to a file. 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

```
pu_tbl <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)

feat_tbl <- data.frame(
  id = 1:2,
  name = c("feature_1", "feature_2")
)

dist_feat_tbl <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)

x <- create_problem(
  pu = pu_tbl,
  features = feat_tbl,
  dist_features = dist_feat_tbl,
  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

Example distribution of feature amounts across planning units.

Usage

```
data(sim_dist_features)
```

Format

A data frame linking planning units and features with an `amount` column.

sim_features *Simulated features*

Description

Example feature table for package examples and tests.

Usage

```
data(sim_features)
```

Format

A data frame with feature identifiers and names.

sim_pu *Simulated planning units*

Description

Example planning units as an `sf` object for package examples and tests.

Usage

```
data(sim_pu)
```

Format

An object of class `sf`.

sim_pu_sf *Simulated planning units*

Description

Example planning units as an `sf` object for package examples and tests.

Usage

```
data(sim_pu_sf)
```

Format

An object of class `sf`.

solution-class *Solution class*

Description

The `Solution` class stores the result of solving a single `Problem` object in **multiscape**.

A `Solution` object contains the original problem definition, the core optimization output returned by the solver, user-facing summary tables, diagnostics about the solve process, and metadata describing how the solution was obtained.

Objects of this class are typically created by `solve`.

Details

Conceptual role

The `Solution` class represents the output of one optimization run.

It should be understood as the single-run counterpart of the modelling workflow:

```
Problem -> solve() -> Solution
```

Thus, a `Solution` object does not replace the original `Problem`; instead, it keeps a reference to it in the `problem` field and augments it with optimization results.

Single-run semantics

A `Solution` corresponds to one realized solution of one configured optimization problem. This may come from:

- a single-objective solve,
- one run of a weighted multi-objective workflow,
- one ϵ-constraint subproblem,

- one AUGMECON subproblem,
- or any other workflow that ultimately produces one optimizer output.

When multiple runs are generated, they are typically collected in a separate `SolutionSet` object rather than a single `Solution`.

Internal structure

A `Solution` object separates results into several layers:

`problem` The original `Problem` object used to generate the solution.

`solution` A named `list` containing the core optimization output. This may include the objective value, raw model variable vector, decoded decision vectors, and evaluated objective values by alias.

`summary` A named `list` of user-facing summary tables, typically derived from the original problem and the solved decisions. These tables are intended for inspection, plotting, reporting, and downstream analysis.

`diagnostics` A named `list` containing solver diagnostics such as status, runtime, optimality gap, solver name, and runtime settings.

`method` A named `list` describing the optimization method used to obtain the solution.

`meta` A named `list` containing additional metadata.

`name` A `character(1)` identifier for the solution object.

Core optimization output

The `solution` field stores the solver-facing result. Typical entries may include:

`objective` The scalar objective value returned by the solver for this run.

`vector` The raw internal solution vector in model-variable order.

`alias_values` Objective values evaluated for registered aliases, when available.

The raw solution vector may contain not only user-facing decision variables such as planning-unit or action decisions, but also auxiliary variables introduced internally by the model builder.

User-facing summaries

The `summary` field stores derived tables intended for interpretation rather than solver interaction. Typical entries include:

`pu` Planning-unit summary table.

`actions` Planning unit–action allocation summary table.

`features` Feature-level outcome summary table.

`targets` Target-achievement summary table, when targets were part of the problem.

These tables are the main source used by user-facing accessors such as `get_pu`, `get_actions`, `get_features`, and `get_targets`.

Diagnostics

The `diagnostics` field stores metadata about the optimization process, including solver status and runtime information. Typical entries may include:

- solver name,
- runtime in seconds,
- optimality gap,
- number of cores,
- time limit,
- status code or status label.

These values describe how the solution was obtained, not the content of the solution itself.

Printing

The `print()` method is intended as a concise diagnostic summary. It reports:

- solver status,
- objective value,
- optimality gap,
- runtime,
- counts of selected planning units and actions,
- target fulfillment summary when available,
- evaluated objective alias values when available,
- and basic solver information.

This summary is intended for quick inspection. More detailed exploration should use the stored `summary`, `solution`, and `diagnostics` fields directly, or the dedicated accessor functions.

Value

No return value. This page documents the `Solution` class.

Fields

`problem` The `Problem` object used to generate the solution.

`solution` A named `list` containing the core optimization result.

`summary` A named `list` of user-facing summary tables.

`diagnostics` A named `list` of solver diagnostics.

`method` A named `list` describing the method used.

`meta` A named `list` of additional metadata.

`name` A `character(1)` name for the solution object.

Methods

`print()` Print a concise summary of the solution, including status, objective value, runtime, selection counts, and target fulfillment.

`show()` Alias of `print()`.

`repr()` Return a short one-line representation of the solution.

See Also

problem-class, get_pu, get_actions, get_features, get_targets, solve

`solutionset-class` *SolutionSet class*

Description

The `SolutionSet` class stores the result of solving a `Problem` object when multiple runs are produced.

A `SolutionSet` object is the multi-run counterpart of `solution-class`. It contains the original problem, run-level design and outcome tables, the list of individual `Solution` objects, diagnostics summarizing the full set of runs, method metadata, and additional metadata.

Objects of this class are typically created by `solve` in workflows that generate more than one optimization run, such as weighted-sum scans, ϵ-constraint designs, or AUGMECON grids.

Details

Conceptual role

The `SolutionSet` class represents the result of a multi-run solving workflow:

```
Problem -> solve() -> SolutionSet
```

Each run corresponds to one specific optimization subproblem, parameter setting, or trade-off configuration. The `SolutionSet` object keeps all these runs together in a structured form.

Thus, a `SolutionSet` is not a single solution with multiple labels, but a collection of distinct `Solution` objects linked to a shared problem and a shared multi-run design.

Typical use cases

A `SolutionSet` is typically returned when the chosen solution method generates multiple runs, for example:

- several weighted-sum configurations,
- a grid of ϵ-constraint runs,
- an AUGMECON exploration of multiple ϵ-combinations,
- or any other workflow producing more than one optimizer call.

Internal structure

A `SolutionSet` object separates information into several layers:

`problem` The original `Problem` object shared by all runs.

`solution` A named `list` containing the core multi-run optimization outputs. This typically includes the run design, the run summary table, and the list of individual `Solution` objects.

`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 multi-run 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 entry point for run-level information. Typical entries include:

`design` A `data.frame` describing the experimental or optimization design, for example weights or ϵ-levels.

`runs` A `data.frame` summarizing the outcome of each run. This typically includes run identifiers, solver status, runtime, gap, and objective values.

`solutions` A list of individual `Solution` objects, one per run.

In many workflows, the `runs` table is the most important compact representation of the solution set, while `solutions[[i]]` provides the full detailed output for run i.

Objective values and run tables

In multi-objective workflows, the run table often stores 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 naming convention is used by downstream functions such as `plot_tradeoff`.

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.

Relationship with `Solution`

A `SolutionSet` is conceptually a collection of `Solution` objects. The individual runs are typically stored in:

$$\texttt{x}solution\texttt{solutions[[i]]}$$

where each element is itself a full `Solution` object.

Therefore:

- use `SolutionSet` when working with the full set of runs,
- use an individual `Solution` when inspecting one particular run in detail.

Diagnostics

The `diagnostics` field stores metadata about the multi-run solve process. Depending on the implementation, it may summarize:

- number of runs,
- 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 full solution set. It reports:

- the optimization method name,
- the participating objective aliases,
- the number of design rows, runs, and stored solutions,
- run-level status summaries,
- runtime and gap ranges,
- and the names of design and objective-value columns when available.

This printed output is intended as a quick overview. Detailed inspection should use:

- `x$solution$runs`,
- `x$solution$design`,
- `x$solution$solutions[[i]]`,
- or the accessor methods documented below.

Value

No return value. This page documents the `SolutionSet` class.

Fields

`problem` The `Problem` object used to generate the full solution set.

`solution` A named `list` containing the core multi-run outputs, typically including `design`, `runs`, and `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 list of individual `Solution` objects stored in `self$solution$solutions`.

See Also

solution-class, plot_tradeoff, get_pu, get_actions, get_features, get_targets, solve

---

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 either a solution-class or a solutionset-class depending on whether the workflow is single-objective or multi-objective.

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 vs multi-objective behaviour

The behaviour of `solve()` depends on the problem configuration.

- **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 `solution-class` object.
- **Multi-objective case.** If a multi-objective method is configured, `solve()` dispatches internally according to the stored method name and returns a `solutionset-class` object.

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.

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

Either:

- a solution-class object when solving a single-objective problem, or
- a solutionset-class object when solving a configured multi-objective problem.

A Solution represents one optimization run. A SolutionSet represents multiple runs together with their run table, design information, and individual Solution objects.

After solving, users will typically inspect or visualize the returned results through methods associated with Solution and SolutionSet objects.

See Also

problem-class, solution-class, solutionset-class, compile_model, set_solver, set_solver_cbc, set_solver_gurobi, set_method_weighted_sum, set_method_epsilon_constraint, set_method_augmecon

Examples

```
# ------------------------------------------------------------
# Minimal single-objective example
# ------------------------------------------------------------
pu <- data.frame(
  id = 1:4,
  cost = c(1, 2, 3, 4)
)

features <- data.frame(
  id = 1:2,
  name = c("sp1", "sp2")
)

dist_features <- data.frame(
  pu = c(1, 1, 2, 3, 4),
  feature = c(1, 2, 2, 1, 2),
  amount = c(5, 2, 3, 4, 1)
)

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

effects <- data.frame(
  action = rep(c("conservation", "restoration"), each = 2),
  feature = rep(features$id, times = 2),
  multiplier = c(0.10, 0.10, 0.50, 0.50)
)

x <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features
```

```
) |>
  add_actions(
    actions = actions,
    cost = c(conservation = 1, restoration = 2)
  ) |>
  add_effects(
    effects = effects,
    effect_type = "after"
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost")

if (requireNamespace("rcbc", quietly = TRUE)) {
  x <- set_solver_cbc(x, verbose = FALSE)
  sol <- solve(x)
  print(sol)
}

# ------------------------------------------------------------
# Minimal multi-objective example
# ------------------------------------------------------------
x_mo <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features
) |>
  add_actions(
    actions = actions,
    cost = c(conservation = 1, restoration = 2)
  ) |>
  add_effects(
    effects = effects,
    effect_type = "after"
  ) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost") |>
  add_objective_max_benefit(alias = "benefit") |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    weights = c(0.5, 0.5),
    normalize_weights = TRUE
  )

if (requireNamespace("rcbc", quietly = TRUE)) {
  x_mo <- set_solver_cbc(x_mo, verbose = FALSE)
  solset <- solve(x_mo)
  print(solset)
}
```

Index