Package: multiscape (via r-universe)

June 7, 2026

Type Package

Version 1.1.0

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, moocore

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' 'frontier.R' 'internalMO.R' 'get_solution.R' 'globals.R' 'load_sim.R' 'mo_control.R' 'package.R' 'plot.R' 'print.R' 'problem-class.R' 'run_design.R' 'selection_analysis.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' 'solution_management.R' 'solve.R' 'utils-pipe.R' 'zzz.R'

Roxygen list(markdown = TRUE)

Config/testthat/edition 3

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

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

Date/Publication 2026-06-07 16:49:22 UTC

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

RemoteRef HEAD

RemoteSha 0d458f3aad70bc12fb79c8253aad5e6d064cc343

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`.

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

---

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 an effects table, with one row per (`pu`, `action`, `feature`) triple and three main effect columns:

- `amount_after`: the feature amount expected after applying the action,
- `benefit`: the positive component of the net change,
- `loss`: the magnitude of the negative component of the net change.

Let i index planning units, a index actions, and f index features. Let b_{if} denote the baseline amount of feature f in planning unit i, and let Δ_{iaf} denote the net effect of applying action a. The after-action amount is:

$$\text{amount\_after}_{iaf} = b_{if} + \Delta_{iaf}.$$

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 stored effect 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 amount_after and 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:

$$\text{amount\_after}_{iaf} = b_{if} + \Delta_{iaf},$$

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

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

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 and `amount_after` equals the baseline amount.

Thus, `benefit` and `loss` describe the net change relative to the baseline, whereas `amount_after` describes the final feature amount under the action. This distinction is important for actions that maintain baseline values. For example, if an action preserves a feature unchanged, then `benefit = 0`, `loss = 0`, and `amount_after` equals the baseline amount.

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. The interpretation depends on `effect_type`.

   If `effect_type = "delta"`, the multiplier represents a relative net change:

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

   If `effect_type = "after"`, the multiplier represents the after-action amount relative to the baseline:

   $$\text{amount\_after}_{iaf} = b_{if} \times m_{af},$$

   and the net effect is:

   $$\Delta_{iaf} = \text{amount\_after}_{iaf} - b_{if} = b_{if}(m_{af} - 1).$$

   Thus, under `effect_type = "after"`, a multiplier of 1 means no change, a multiplier below 1 means a loss, and a multiplier above 1 means a gain. 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 and requiring `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. For explicit `delta` or `effect` columns, values are used as signed changes. For `multiplier` inputs, values are interpreted as relative net changes:

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

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

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

For `multiplier` inputs under `effect_type = "after"`, the after-action amount is computed as $b_{if} \times m_{af}$, so that:

$$\Delta_{iaf} = b_{if}(m_{af} - 1).$$

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 stored effect rows, including neutral effects,
- `component = "benefit"`: keep only rows with `benefit > 0`,
- `component = "loss"`: keep only rows with `loss > 0`.

Zero-effect rows are retained by default because they may encode valid neutral effects. They are removed only when using `component = "benefit"` or `component = "loss"`.

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`, `amount_after`, `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`

`add_losses` *Add losses*

Description

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

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

---

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)
```

`frontier_distances` *Compute distances to observed ideal or nadir points*

Description

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

Usage

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

Arguments

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

Details

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

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

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

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

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

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

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

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

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

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

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

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

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

After normalization:

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

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

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

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

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

Value

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

The table contains:

- run_id and solution_id;
- the original objective values;
- normalized objective values prefixed with norm_;
- distance and rank columns for the requested reference points.

The returned table also contains the following attributes:

- "ideal": observed ideal point in the original objective scales;

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

See Also

frontier_extremes, get_objectives, get_objective_specs, solution_filter

Examples

```
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(actions$id, each = 2),
  feature = rep(features$id, times = 2),
  multiplier = c(
    1.0, 1.0,
    1.5, 1.5
  )
)

problem <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
```

```
  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"),
    runs = run_grid(
      n = 5,
      include_extremes = TRUE
    ),
    normalize_weights = TRUE
  )

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

  solutions <- solve(problem)

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

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

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

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

  # Use Chebyshev distance
```

```
  frontier_distances(
    solutions,
    metric = "chebyshev"
  )

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

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

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

`frontier_extremes` *Find objective-wise extreme solutions*

Description

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

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

Usage

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

Arguments

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

Details

Objective values are obtained from `get_objectives` with `format = "wide"`. Objective senses are obtained from `get_objective_specs`.

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

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

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

Value

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

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

See Also

`get_objectives`, `get_objective_specs`, `solution_filter`

Examples

```
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(actions$id, each = 2),
  feature = rep(features$id, times = 2),
  multiplier = c(
    1.0, 1.0,
    1.5, 1.5
  )
)

problem <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  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"),
    runs = run_grid(
      n = 5,
      include_extremes = TRUE
    ),
    normalize_weights = TRUE
  )

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

  solutions <- solve(problem)

  # Observed minimum and maximum for every objective
  frontier_extremes(solutions)

  # Inspect only selected objectives
  frontier_extremes(
    solutions,
```

```
    objectives = c("cost", "benefit")
  )

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

`get_actions` *Get action results from a solution set*

Description

Extract the action-allocation summary table from a `solutionset-class` object returned by `solve`.

The returned table summarizes solution values at the planning unit–action level and typically includes a `selected` indicator showing whether each feasible `(pu, action)` pair is selected in a run.

Usage

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

Arguments

`x`	A `solutionset-class` 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. 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 `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

```
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(actions$id, each = 2),
  feature = rep(features$id, times = 2),
  multiplier = c(
    1.0, 1.0,
    1.5, 1.5
  )
)

problem <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  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)) {
  problem <- set_solver_cbc(
    problem,
    verbose = FALSE
  )

  solutions <- solve(problem)

  # All feasible planning-unit/action assignments
  get_actions(solutions)

  # Only selected action assignments
  get_actions(
    solutions,
    only_selected = TRUE
  )

  # Action allocations for one run
  run_ids <- get_runs(solutions)$run_id

  get_actions(
    solutions,
    run = run_ids[1]
  )
}
```

`get_features` *Get feature summary from a solution set*

Description

Extract the per-feature summary table from a `solutionset-class` object returned by `solve`.

The returned table summarizes, for each feature, how much of the feature was available in the full baseline landscape and how much is represented by the selected planning units or selected actions in each run.

Usage

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

Arguments

x	A solutionset-class object returned by solve.
run	Optional positive integer giving the run index to extract. 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.

Feature summaries distinguish between baseline availability in the full landscape and the contribution of selected planning units or selected actions.

Let B_f denote the total baseline amount of feature f available in the full landscape. Let S_f denote the baseline amount of feature f in selected rows. Let A_f denote the after-action amount of feature f contributed by those selected rows. Let G_f and L_f denote the positive and negative net-change components induced by selected actions. Then:

$$\text{selected\_net}_f = G_f - L_f,$$

and:

$$A_f = S_f + \text{selected\_net}_f.$$

The main returned columns are:

- baseline_total: total baseline amount in the full landscape;
- selected_baseline: baseline amount in selected rows;
- selected_amount_after: after-action amount contributed by selected rows;
- selected_benefit: positive net-change component from selected actions;
- selected_loss: negative net-change component from selected actions;
- selected_net: net change from selected actions;
- selected_fraction_of_baseline: ratio between selected_amount_after and baseline_total.

Importantly, this summary does not assume that planning units without a selected action contribute to the achieved feature amount. Therefore, the achieved amount for a feature is represented by selected_amount_after, not by a full-landscape total obtained by adding net changes to the baseline.

For backwards compatibility with older result objects, if the newer selected-action columns are missing, this function attempts to construct them from older columns such as total_available, benefit, loss, net, and amount_after. However, the returned table is organized using the newer selected-action terminology.

If 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 result. It is different from get_targets, which focuses on target achievement.

Value

A `data.frame` with one row per feature, or one row per feature–run combination when multiple runs are present. The returned table includes, when available or derivable, the columns `baseline_total`, `selected_baseline`, `selected_amount_after`, `selected_benefit`, `selected_loss`, `selected_net`, and `selected_fraction_of_baseline`.

See Also

`get_pu`, `get_actions`, `get_targets`

Examples

```
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)
)

problem <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost")

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

  solutions <- solve(problem)

  # Feature outcomes for all stored runs
  get_features(solutions)

  # Feature outcomes for one run
  run_ids <- get_runs(solutions)$run_id

  get_features(
```

```
    solutions,
    run = run_ids[1]
  )
}
```

`get_objective_specs` *Get objective specifications from a solution set*

Description

Extract the definitions of the objectives registered in the original planning problem associated with a `solutionset-class` object.

Usage

```
get_objective_specs(x)
```

Arguments

x	A `solutionset-class` object returned by `solve`.

Details

Objective specifications are read from `x$problem$data$objectives`. They describe how each objective was registered, independently of the multi-objective method later used to solve the problem.

The returned optimization sense is used by frontier and dominance functions to place objectives in a common minimization space.

Value

A `data.frame` with one row per registered objective and the columns:

- `objective`: user-defined objective alias;
- `objective_id`: internal objective type;
- `model_type`: internal model formulation;
- `sense`: optimization direction, `"min"` or `"max"`;
- `created_at`: objective registration timestamp.

See Also

`get_runs`, `get_objectives`, `frontier_extremes`, `solution_filter`

Examples

```
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(actions$id, each = 2),
  feature = rep(features$id, times = 2),
  multiplier = c(
    1.0, 1.0,
    1.5, 1.5
  )
)

problem <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  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"),
```

```
    runs = run_grid(
      n = 5,
      include_extremes = TRUE
    ),
    normalize_weights = TRUE
  )

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

  solutions <- solve(problem)

  get_objective_specs(solutions)
}
```

`get_objectives` *Get objective values from a solution set*

Description

Extract objective values from the runs stored in a `solutionset-class` object.

Usage

```
get_objectives(x, format = c("long", "wide"), feasible_only = FALSE)
```

Arguments

`x`	A `solutionset-class` object returned by `solve`.
`format`	Character. Output representation, either `"long"` or `"wide"`. Defaults to `"long"`.
`feasible_only`	Logical. If `TRUE`, extract values only from runs whose status is interpreted as usable. Defaults to `FALSE`.

Details

Objective values are read from run-table columns named `value_<objective>`, where `<objective>` is the registered objective alias.

Runs without a stored solution may contain missing objective values. Use `feasible_only = TRUE`, or filter the `SolutionSet` beforehand, when only solved runs should be included.

In long format, every run-objective combination occupies one row. In wide format, every run occupies one row and every objective occupies one column.

Value

If `format = "long"`, a `data.frame` with columns `run_id`, `solution_id`, `objective`, and `value`.

If `format = "wide"`, a `data.frame` with `run_id`, `solution_id`, and one column per objective.

See Also

`get_runs`, `get_objective_specs`, `frontier_extremes`, `frontier_distances`

Examples

```
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(actions$id, each = 2),
  feature = rep(features$id, times = 2),
  multiplier = c(
    1.0, 1.0,
    1.5, 1.5
  )
)

problem <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  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"),
    runs = run_grid(
      n = 5,
      include_extremes = TRUE
    ),
    normalize_weights = TRUE
  )

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

  solutions <- solve(problem)

  # Long format
  get_objectives(solutions)

  # Wide format
  get_objectives(
    solutions,
    format = "wide"
  )

  # Objective values from usable runs only
  get_objectives(
    solutions,
    feasible_only = TRUE
  )
}
```

`get_pu` *Get planning-unit results from a solution set*

Description

Extract the planning-unit summary table from a `solutionset-class` object returned by `solve`.

The returned table summarizes solution values at the planning-unit level and typically includes a `selected` indicator showing whether each planning unit is selected in a run.

Usage

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

Arguments

x	A solutionset-class 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. 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 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

```
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)
)

problem <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost")

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

  solutions <- solve(problem)

  # Planning-unit results for all stored runs
  get_pu(solutions)

  # Return only selected planning units
  get_pu(
    solutions,
    only_selected = TRUE
  )

  # Extract one run using its run_id
  run_ids <- get_runs(solutions)$run_id

  get_pu(
    solutions,
    run = run_ids[1]
  )
}
```

`get_runs` *Get run-level results from a solution set*

Description

Extract the run table from a `solutionset-class` object.

Usage

```
get_runs(x, feasible_only = FALSE)
```

Arguments

x	A solutionset-class object returned by solve.
feasible_only	Logical. If TRUE, return only runs whose status indicates that a usable solution may be available. Defaults to FALSE.

Details

A run represents an attempted optimization solve. Each run has a unique run_id, but only runs that produce a stored solution receive a solution_id.

Consequently, the number of runs may exceed the number of stored solutions. This commonly occurs when a multi-objective design contains infeasible, failed, or interrupted runs.

The run table combines:

- run and solution identifiers;
- solver status, runtime, gap, and messages;
- multi-objective design parameters such as weights or epsilon levels;
- objective values stored in columns named value_<objective>.

If feasible_only = TRUE, runs with statuses "optimal", "feasible", "suboptimal", "time_limit", or "gap_limit" are retained.

Value

A data.frame with one row per attempted optimization run.

See Also

get_objectives, get_objective_specs, solution_filter, run_grid, run_manual

Examples

```
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(actions$id, each = 2),
  feature = rep(features$id, times = 2),
  multiplier = c(
    1.0, 1.0,
    1.5, 1.5
  )
)

problem <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  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"),
    runs = run_grid(
      n = 5,
      include_extremes = TRUE
    ),
    normalize_weights = TRUE
  )

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

  solutions <- solve(problem)

  # All attempted runs
  get_runs(solutions)

  # Only runs with a usable solver status
  get_runs(
```

```
    solutions,
    feasible_only = TRUE
  )
}
```

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

Description

Return the raw decision-variable vector for a selected run in a `solutionset-class` object returned by `solve`.

The vector is returned in the internal model-variable order used by the optimization backend.

Usage

```
get_solution_vector(x, run = NULL, solution_id = NULL)
```

Arguments

`x`	A `solutionset-class` object returned by `solve`.
`run`	Optional positive integer giving the run id to extract. If `NULL`, the first stored solution is used unless `solution_id` is supplied.
`solution_id`	Optional character string giving the solution id to extract. If supplied, `run` must be `NULL`.

Details

This function extracts the raw decision vector for one run. 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.

Value

A numeric vector with one value per internal model variable.

See Also

get_pu, get_actions, get_features, get_targets

Examples

```
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)
)

problem <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost")

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

  solutions <- solve(problem)

  # Extract the first stored raw solution vector
  vector <- get_solution_vector(solutions)
  vector
  length(vector)

  # Extract a vector using its solution_id
  runs <- get_runs(solutions)
  solution_ids <- runs$solution_id[
    !is.na(runs$solution_id)
  ]

  if (length(solution_ids) > 0L) {
    get_solution_vector(
      solutions,
```

```
      solution_id = solution_ids[1]
    )
  }
}
```

`get_targets` *Get target achievement summary from a solution set*

Description

Extract a user-facing target-achievement table from a `solutionset-class` object returned by `solve`.

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

Usage

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

Arguments

`x`	A `solutionset-class` object returned by `solve`.
`run`	Optional positive integer giving the run index to extract. If `NULL`, all runs are returned when available.

Details

Targets are optional in **multiscape**. If the result object does not contain a targets summary table at `x$summary$targets`, this function returns `NULL` without error.

This function reads the stored targets summary and returns a simplified user-facing table. If the summary contains `achieved` and `target_value`, target satisfaction is evaluated as follows.

For lower-bound targets:

$$\text{met} = (\text{achieved} \geq \text{target}),$$

and for upper-bound targets:

$$\text{met} = (\text{achieved} \leq \text{target}).$$

The interpretation of the target direction is taken from the `sense` column when available:

- `"ge"`, `">="`, or `"min"` are treated as lower-bound targets;
- `"le"`, `"<="`, or `"max"` are treated as upper-bound targets;
- if `sense` is missing, the target is treated as a lower bound by default.

The returned table is simplified and renames some internal fields for readability:

- `target_raw` is returned as `target_level`;
- `basis_total` is returned as `total_available`;

- `target_value` is returned as `target`.

If `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 result 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

```
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)
)

problem <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost")

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

```
  solutions <- solve(problem)

  # Target requirements and achieved amounts
  get_targets(solutions)

  # Target achievement for one run
  run_ids <- get_runs(solutions)$run_id

  get_targets(
    solutions,
    run = run_ids[1]
  )
}
```

`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`.

`mo_control` *Control multi-objective method behavior*

Description

Create a control object that determines how multi-objective workflows respond to infeasible runs, missing solution vectors, unexpected errors, and AUGMECON slack-variable bounds.

The resulting object is supplied to the `control` argument of `set_method_epsilon_constraint` or `set_method_augmecon`.

Usage

```
mo_control(
  stop_on_infeasible = FALSE,
  stop_on_no_solution = FALSE,
  stop_on_error = TRUE,
  slack_upper_bound = 1e+06
)
```

Arguments

`stop_on_infeasible`
: Logical. If `TRUE`, stop the complete multi-objective workflow when a run is reported as infeasible. If `FALSE`, retain the attempted run in the run table and continue with the remaining runs. Defaults to `FALSE`.

`stop_on_no_solution`
: Logical. If `TRUE`, stop when a run does not return a usable solution vector. If `FALSE`, retain the attempted run without a stored `solution_id` and continue. Defaults to `FALSE`.

`stop_on_error`
: Logical. If `TRUE`, stop on unexpected errors raised during model preparation, solving, or result processing. If `FALSE`, attempt to record the failed run and continue. Defaults to `TRUE`.

`slack_upper_bound`
: A single positive finite numeric value defining the upper bound of AUGMECON slack variables. This setting is ignored by other multi-objective methods. Defaults to `1e6`.

Details

Multi-objective methods commonly solve a sequence of related optimization models. Some parameter combinations, particularly restrictive epsilon levels, may be infeasible or may fail to produce a usable solution.

`mo_control()` determines whether such events stop the entire multi-objective workflow or are recorded in the resulting `solutionset-class` object.

A `SolutionSet` distinguishes between:

- a `run_id`, which identifies every attempted optimization run;
- a `solution_id`, which is assigned only when a run produces a stored solution.

When a run is retained after an infeasibility or missing-solution event, its run-level metadata remains available through `get_runs`, but its `solution_id` and objective values will generally be missing.

Infeasible runs

If `stop_on_infeasible = FALSE`, an infeasible run is recorded and the workflow continues with the remaining run design. This is generally useful when exploring automatically generated epsilon grids because restrictive combinations of epsilon levels may be infeasible.

If `stop_on_infeasible = TRUE`, the workflow stops when the first infeasible run is encountered.

Runs without a solution vector

A solver may terminate without returning a usable decision vector even when the outcome is not classified explicitly as infeasible.

If `stop_on_no_solution = FALSE`, the attempted run is recorded without a stored solution and the remaining runs are attempted. If `stop_on_no_solution = TRUE`, the workflow stops immediately.

Unexpected errors

If `stop_on_error = TRUE`, unexpected errors raised while preparing, solving, or processing a run are propagated and stop the workflow. This is the recommended default because such errors may indicate an invalid model, unsupported solver behaviour, or an internal implementation problem.

If `stop_on_error = FALSE`, the workflow attempts to record the failed run and continue. This option should be used cautiously because it may conceal modelling or implementation errors.

AUGMECON slack upper bound

`slack_upper_bound` defines an explicit upper bound for slack variables introduced by the AUGMECON formulation. It is used only by `set_method_augmecon` and has no effect on weighted-sum or standard epsilon-constraint workflows.

The value should be sufficiently large to avoid excluding valid solutions, but unnecessarily large bounds can weaken the mixed-integer formulation and reduce numerical performance. When possible, a problem-specific bound based on the ranges of the constrained objectives should be used.

The default settings favour completing the requested run design while still stopping on unexpected errors:

```
stop_on_infeasible = FALSE
stop_on_no_solution = FALSE
stop_on_error = TRUE
slack_upper_bound = 1e6
```

Value

An object of class `MOControl` containing the validated execution-control settings. The object is intended to be supplied to the `control` argument of a supported multi-objective method.

See Also

`set_method_epsilon_constraint`, `set_method_augmecon`, `run_grid`, `run_manual`, `get_runs`, `solutionset-class`

Examples

```
# Default behaviour: continue after infeasible runs or runs without a
# solution, but stop on unexpected errors
control <- mo_control()
control

# Stop as soon as an infeasible run or missing solution is encountered
strict_control <- mo_control(
  stop_on_infeasible = TRUE,
  stop_on_no_solution = TRUE
)

strict_control

# Define a problem with explicit conservation and restoration actions
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(actions$id, each = 2),
  feature = rep(features$id, times = 2),
  multiplier = c(
    1.0, 1.0,
    1.5, 1.5
  )
)

problem <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  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_epsilon_constraint(
    primary = "cost",
    runs = run_grid(
      n = 5,
      include_extremes = TRUE
    ),
    control = mo_control(
      stop_on_infeasible = FALSE,
      stop_on_no_solution = FALSE,
      stop_on_error = TRUE
    )
```

```
  )

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

  solutions <- solve(problem)

  # All attempted runs, including runs without stored solutions
  get_runs(solutions)

  # Only runs with a usable solver status
  get_runs(
    solutions,
    feasible_only = TRUE
  )
}
```

---

plot_spatial *Plot spatial outputs from a solution set*

Description

Convenience wrapper to plot spatial outputs from a solutionset-class object returned by solve.

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 solutionset-class object returned by solve.
what	Character string indicating what to plot. Must be one of "pu", "actions", or "features".
runs	Optional integer vector of run ids. If NULL, the first available run is plotted by default.
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) &&
  requireNamespace("rcbc", quietly = TRUE)
) {
  data("sim_pu_sf", package = "multiscape")

  pu <- sim_pu_sf[
    seq_len(min(4L, nrow(sim_pu_sf))),
  ]

  pu$id <- seq_len(nrow(pu))
  pu$cost <- seq_len(nrow(pu))

  features <- data.frame(
    id = 1L,
    name = "feature_1"
  )

  dist_features <- data.frame(
    pu = pu$id,
    feature = 1L,
    amount = rep(1, nrow(pu))
  )

  problem <- create_problem(
    pu = pu,
    features = features,
    dist_features = dist_features,
    cost = "cost"
  ) |>
    add_constraint_targets_relative(0.25) |>
    add_objective_min_cost(alias = "cost") |>
    set_solver_cbc(verbose = FALSE)

  solutions <- solve(problem)

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

`plot_spatial_actions` *Plot selected actions in space*

Description

Plot the spatial distribution of selected actions from a `solutionset-class` object returned by `solve`.

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

Usage

```
plot_spatial_actions(
  x,
  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 `solutionset-class` object returned by `solve`.
`runs`	Optional integer vector of run ids. If `NULL`, the first available run is plotted by default.
`actions`	Optional action subset to display. Entries may match action ids or action-set labels.
`layout`	Character string controlling the layout. Must be one of `"single"` or `"facet"`. If `NULL`, the default is `"single"`.
`max_facets`	Maximum number of action facets shown when `actions` is `NULL` and faceting would otherwise create many panels.
`...`	Reserved for future extensions.
`base_alpha`	Numeric value in $[0, 1]$ giving the alpha of the base planning-unit layer.

selected_alpha	Numeric value in $[0, 1]$ giving the alpha of the highlighted action layer.
base_fill	Fill colour for the base planning-unit layer.
base_color	Border colour for the base planning-unit layer.
selected_color	Border colour for highlighted layers.
draw_borders	Logical. If FALSE, borders are not drawn.
show_base	Logical. If TRUE, draw the base planning-unit layer underneath the highlighted output.
fill_values	Optional named vector of colours for discrete action maps.
fill_na	Fill colour for missing values.
use_viridis	Logical. If TRUE and the **viridis** package is available, use viridis discrete scales.

Details

Let $x_{ia} \in \{0, 1\}$ denote whether action a is selected in planning unit i. This function plots the selected (pu, action) pairs in geographic space.

If layout = "facet" and only one run is plotted, one panel is drawn per action.

If layout = "single", all selected actions are drawn in a single map using discrete fills. If more than one action is selected in the same planning unit, the action labels are collapsed using "+".

When plotting multiple runs, only layout = "single" is supported.

Planning-unit geometry must be available in the associated problem object.

Value

Invisibly returns a ggplot object.

See Also

get_actions, plot_spatial, plot_spatial_pu, plot_spatial_features

Examples

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

  pu <- sim_pu_sf[
    seq_len(min(4L, nrow(sim_pu_sf))),
  ]

  pu$id <- seq_len(nrow(pu))
  pu$cost <- seq_len(nrow(pu))

  features <- data.frame(
    id = 1L,
```

```
  name = "feature_1"
)

dist_features <- data.frame(
  pu = pu$id,
  feature = 1L,
  amount = rep(1, nrow(pu))
)

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

effects <- data.frame(
  action = actions$id,
  feature = 1L,
  multiplier = c(1.0, 1.5)
)

problem <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  add_actions(
    actions = actions,
    cost = c(
      conservation = 1,
      restoration = 2
    )
  ) |>
  add_effects(
    effects = effects,
    effect_type = "after"
  ) |>
  add_constraint_targets_relative(0.25) |>
  add_objective_min_cost(alias = "cost") |>
  set_solver_cbc(verbose = FALSE)

solutions <- solve(problem)

plot_spatial_actions(
  solutions,
  layout = "single"
)
}
```

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

Description

Plot feature values in space from a solutionset-class object returned by solve.

This function combines baseline feature amounts from the associated Problem object with positive effects induced by the actions selected in each stored run to produce planning-unit-level feature maps. Selected actions are obtained through get_actions.

Usage

```
plot_spatial_features(
  x,
  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 solutionset-class object returned by solve.
runs	Optional integer vector of run ids. If NULL, the first available run is plotted by default.
features	Optional feature subset to display. Matching is attempted against both feature ids and feature names.
value	Character string indicating which feature quantity to plot. Must be one of "final", "baseline", or "benefit".
layout	Character string controlling the layout. Must be one of "single" or "facet". If NULL, the default is "facet".
max_facets	Maximum number of feature facets shown when features = NULL and faceting would otherwise create many panels.
...	Reserved for future extensions.
base_alpha	Unused in the current feature view, kept for interface consistency.
selected_alpha	Unused in the current feature view, kept for interface consistency.
base_fill	Unused in the current feature view, kept for interface consistency.
base_color	Unused in the current feature view, kept for interface consistency.

selected_color	Border colour for filled feature polygons.
draw_borders	Logical. If FALSE, borders are not drawn.
show_base	Unused in the current feature view, kept for interface consistency.
fill_na	Fill colour for missing values.
use_viridis	Logical. If TRUE and the **viridis** package is available, use a continuous viridis scale.

Details

For each planning unit i and feature f, the plotted quantities are:

$$\text{baseline}_{if},$$

$$\text{benefit}_{if},$$

$$\text{final}_{if} = \text{baseline}_{if} + \text{benefit}_{if}.$$

In the current implementation:

- baseline is the summed baseline amount from dist_features;
- benefit is the summed positive effect from selected actions;
- final is baseline + benefit.

Negative effects are not subtracted in this plotting method. Therefore, value = "final" should be interpreted as baseline plus selected positive effects under the current plotting logic.

If layout = "facet" and only one run is plotted, one panel is drawn per feature.

If multiple runs are plotted, exactly one feature must be requested, and faceting is done by run.

Planning-unit geometry must be available in the associated problem object.

Value

Invisibly returns a ggplot object.

See Also

get_features, plot_spatial, plot_spatial_pu, plot_spatial_actions

Examples

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

  pu <- sim_pu_sf[
    seq_len(min(4L, nrow(sim_pu_sf))),
  ]
```

```
pu$id <- seq_len(nrow(pu))
pu$cost <- seq_len(nrow(pu))

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

dist_features <- data.frame(
  pu = rep(pu$id, each = 2),
  feature = rep(features$id, times = nrow(pu)),
  amount = c(
    4, 1,
    3, 2,
    2, 3,
    1, 4
  )
)

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

effects <- data.frame(
  action = rep(actions$id, each = 2),
  feature = rep(features$id, times = 2),
  multiplier = c(
    1.0, 1.0,
    1.5, 1.5
  )
)

problem <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  add_actions(
    actions = actions,
    cost = c(
      conservation = 1,
      restoration = 2
    )
  ) |>
  add_effects(
    effects = effects,
    effect_type = "after"
  ) |>
  add_constraint_targets_relative(0.25) |>
  add_objective_min_cost(alias = "cost") |>
  set_solver_cbc(verbose = FALSE)
```

```
  solutions <- solve(problem)

  plot_spatial_features(
    solutions,
    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 `solutionset-class` object returned by `solve`.

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 `solutionset-class` object returned by `solve`.
`runs`	Optional integer vector of run ids. If `NULL`, the first available run is plotted by default.
`...`	Reserved for future extensions.
`base_alpha`	Numeric value in $[0, 1]$ giving the alpha of the base planning-unit layer.
`selected_alpha`	Numeric value in $[0, 1]$ giving the alpha of the selected planning-unit layer.
`base_fill`	Fill colour for the base planning-unit layer.
`base_color`	Border colour for the base planning-unit layer.

selected_color	Border colour for selected planning units.
draw_borders	Logical. If FALSE, borders are not drawn.
show_base	Logical. If TRUE, draw the base planning-unit layer underneath the selected units.

Details

Let $w_i \in \{0, 1\}$ denote the planning-unit selection variable for planning unit i. This function plots the user-facing selected == 1 representation of w_i.

If several runs are requested, the output is faceted by run_id.

Planning-unit geometry must be available in the associated problem object.

Value

Invisibly returns a ggplot object.

See Also

get_pu, plot_spatial, plot_spatial_actions, plot_spatial_features

Examples

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

  pu <- sim_pu_sf[
    seq_len(min(4L, nrow(sim_pu_sf))),
  ]

  pu$id <- seq_len(nrow(pu))
  pu$cost <- seq_len(nrow(pu))

  features <- data.frame(
    id = 1L,
    name = "feature_1"
  )

  dist_features <- data.frame(
    pu = pu$id,
    feature = 1L,
    amount = rep(1, nrow(pu))
  )

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

```
    cost = "cost"
  ) |>
    add_constraint_targets_relative(0.25) |>
    add_objective_min_cost(alias = "cost") |>
    set_solver_cbc(verbose = FALSE)

  solutions <- solve(problem)

  plot_spatial_pu(solutions)
}
```

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

Description

Plot pairwise trade-offs among objective values stored in a `solutionset-class` object.

This function is intended for workflows in which the solution set contains one row per run and two or more objective-value columns of the form `value_*`.

If exactly two objectives are selected, the function returns a single scatterplot. If three or more objectives are selected, all pairwise combinations are plotted using facets.

Usage

```
plot_tradeoff(
  x,
  objectives = NULL,
  color_by = NULL,
  all_pairs = NULL,
  connect = FALSE,
  label_runs = FALSE,
  point_size = 3,
  line_alpha = 0.5,
  text_size = 3,
  ...
)
```

Arguments

`x`	A `solutionset-class` object.
`objectives`	Optional character vector of objective aliases to display. These must match the suffixes of the `value_*` columns in `x$solution$runs`. If `NULL`, all available objective columns are used.
`color_by`	Optional character scalar used to colour points. This may be either one of the selected objective aliases or one of the run-level columns `"run_id"`, `"status"`, `"runtime"`, or `"gap"`.

all_pairs	Logical. If TRUE, allow plotting all pairwise combinations even when more than four objectives are selected. If NULL, it is treated as FALSE.
connect	Logical. If TRUE, connect points by run order within each panel.
label_runs	Logical. If TRUE, add run labels to points.
point_size	Numeric point size.
line_alpha	Numeric alpha value for connecting lines.
text_size	Numeric size for run labels.
...	Reserved for future extensions.

Details

This function reads the run-level table stored in x$solution$runs. It expects objective values to be stored in columns whose names begin with "value_".

If the available objective columns are, for example, value_cost, value_benefit, and value_frag, then the corresponding objective aliases are "cost", "benefit", and "frag".

Let $f_k(r)$ denote the value of objective k in run r. This function visualizes pairwise projections of the run table of the form:

$$(f_k(r), f_\ell(r))$$

for selected pairs of objectives k, ℓ.

If exactly two objectives are selected, a single panel is produced.

If three or more objectives are selected, all pairwise combinations are generated:

$$\{(k, \ell) : k < \ell,\ k, \ell \in \mathcal{O}\},$$

where $\mathcal{O}$ is the selected set of objective aliases.

By default, plotting more than four objectives is not allowed unless all_pairs = TRUE, because the number of panels grows quadratically in the number of objectives.

Colouring

If color_by is supplied, points are coloured by either:

- one of the selected objective aliases, in which case the corresponding value_* column is used;
- or one of the run-level columns run_id, status, runtime, or gap.

Connecting runs

If connect = TRUE, runs are connected in their current table order within each panel. This can be useful when runs correspond to an ordered scan of weights, ϵ-levels, or frontier points, but it should be used with care when run order has no substantive meaning.

Run labels

If label_runs = TRUE, each point is labelled by its run_id. If the **ggrepel** package is available, repelled labels are used.

Value

Invisibly returns a ggplot object.

See Also

solve, solutionset-class

Examples

```
if (
  requireNamespace("ggplot2", quietly = TRUE) &&
  requireNamespace("rcbc", quietly = TRUE)
) {
  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(actions$id, each = 2),
    feature = rep(features$id, times = 2),
    multiplier = c(
      1.0, 1.0,
      1.5, 1.5
    )
  )

  problem <- create_problem(
    pu = pu,
    features = features,
    dist_features = dist_features,
    cost = "cost"
  ) |>
    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"),
      runs = run_grid(
        n = 3,
        include_extremes = TRUE
      ),
      normalize_weights = TRUE
    ) |>
    set_solver_cbc(verbose = FALSE)

  solutions <- solve(problem)

  plot_tradeoff(
    solutions,
    objectives = c("cost", "benefit")
  )
}
```

`problem-class` *Problem class*

Description

The `Problem` class is the central container used by **multiscape** to represent a planning problem before, during, and after model construction.

A `Problem` object stores the full problem specification in a modular way. This includes the baseline planning data, optional spatial metadata, action definitions, effects, profit, targets, constraints, objective registrations, solver settings, and, when available, a built optimization model or model snapshot.

In other words, `Problem` is the persistent object that connects the full `multiscape` workflow:

```
create_problem()
-> add_*() / set_*()
-> solve()
```

Details

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`

`run_grid` *Define an automatic multi-objective run grid*

Description

Create an automatic run-design specification for generating multiple optimization runs in a multi-objective workflow.

`run_grid()` provides a common interface for controlling the resolution of weighted-sum, epsilon-constraint, and AUGMECON run designs. The returned object does not contain the final run table. Instead, it stores the requested grid settings, which are resolved later by the corresponding `set_method_*()` function using the registered objectives and their optimization senses.

Usage

```
run_grid(n, include_extremes = TRUE)
```

Arguments

`n`	Integer. Resolution of the automatically generated run design. Must be at least `2`. The final number of optimization runs may differ from `n`, depending on the selected method and the number of objectives.
`include_extremes`	Logical. Whether objective-space extreme settings should be included in the generated design. Defaults to `TRUE`.

Details

Multi-objective methods generally require several optimization runs to explore different regions of objective space. `run_grid()` asks `multiscape` to generate those runs automatically.

The interpretation of the grid depends on the selected method:

- In `set_method_weighted_sum`, the grid defines combinations of objective weights. The generated weights are normalized according to the method settings and represent alternative preferences among the registered objectives.
- In `set_method_epsilon_constraint`, the grid defines epsilon levels for the constrained objectives. The primary objective is optimized directly, while the remaining objectives are progressively restricted across the generated runs.
- In `set_method_augmecon`, the grid similarly defines epsilon levels for the secondary objectives, which are then used in the augmented epsilon-constraint formulation.

The argument `n` controls the resolution of the automatically generated design. It should not always be interpreted as the final number of runs. For example, when several secondary objectives are present, epsilon levels may be combined across objectives and generate more runs than the value supplied to `n`. Likewise, the number of valid weighted combinations depends on the number of objectives and on how the weight grid is constructed.

If `include_extremes = TRUE`, the generated design includes settings corresponding to the objective-space extremes whenever they are meaningful for the selected method. For weighted-sum methods, this includes weight combinations that place all weight on a single objective. For epsilon-based methods, it includes the boundary levels of the automatically derived objective ranges.

Including extremes is generally recommended because it helps recover the best observed value of each objective and provides reference points for subsequent frontier analyses. However, users may set `include_extremes = FALSE` when only interior trade-off solutions are required or when extreme solutions have already been obtained separately.

The resolved design is stored in the resulting `solutionset-class` object and can be inspected after solving through `get_runs` or the internal run-design table.

Use `run_manual` instead when exact weights or epsilon levels must be supplied explicitly.

Value

An object of class `RunGrid` and `RunDesign`. The object stores the requested grid resolution and extreme-point setting and is intended to be supplied to the `runs` argument of a multi-objective method function.

See Also

`run_manual`, `set_method_weighted_sum`, `set_method_epsilon_constraint`, `set_method_augmecon`, `get_runs`

Examples

```
# Create an automatic run-grid specification
grid <- run_grid(
  n = 5,
```

```
  include_extremes = TRUE
)

grid

# Use the automatic grid in a weighted-sum workflow
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(actions$id, each = 2),
  feature = rep(features$id, times = 2),
  multiplier = c(
    1.0, 1.0,
    1.5, 1.5
  )
)

problem <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  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"),
    runs = run_grid(
      n = 5,
      include_extremes = TRUE
    ),
    normalize_weights = TRUE
  )

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

  solutions <- solve(problem)

  # Inspect the resolved run design and objective values
  get_runs(solutions)
  get_objectives(
    solutions,
    format = "wide"
  )
}
```

`run_manual` *Define a manual multi-objective run design*

Description

Create an explicit run-design specification in which each row defines one multi-objective optimization run.

`run_manual()` is used when exact objective weights or epsilon levels should be supplied by the user instead of being generated automatically with `run_grid`.

Usage

```
run_manual(x)
```

Arguments

`x`	A non-empty `data.frame` with one row per optimization run. Run-design columns must be named using the `weight_<alias>` or `eps_<alias>` convention.

Details

The input must be a non-empty `data.frame` with one row per requested optimization run.

Weighted-sum columns must follow the convention:

`weight_<alias>`

Epsilon-constraint and AUGMECON columns must follow the convention:

`eps_<alias>`

A manual design must contain at least one column beginning with `weight_` or `eps_`. Mixing both column families in the same design is not allowed.

All run-design columns must be numeric, finite, and free of missing values. Weight columns must contain non-negative values, and each weighted-sum row must assign a strictly positive total weight.

This function performs method-independent structural validation. Method-specific validation is performed later by the corresponding `set_method_*()` function. This includes checking:

- that all required objective aliases are represented;
- that no unknown objective columns are supplied;
- that epsilon columns correspond only to secondary objectives;
- and that the supplied design is compatible with the selected method.

Therefore, an object may be structurally valid for `run_manual()` but rejected later by `set_method_weighted_sum`, `set_method_epsilon_constraint`, or `set_method_augmecon`.

Value

An object of class `RunManual` and `RunDesign` containing the validated run table.

See Also

`run_grid`, `set_method_weighted_sum`, `set_method_epsilon_constraint`, `set_method_augmecon`

Examples

```
weighted_runs <- run_manual(
  data.frame(
    weight_cost = c(1.0, 0.5, 0.0),
    weight_benefit = c(0.0, 0.5, 1.0)
  )
)

epsilon_runs <- run_manual(
  data.frame(
    eps_benefit = c(2, 4, 6, 8)
  )
)
```

```
weighted_runs
epsilon_runs
```

`selection_frequency` *Calculate selection frequency across solutions*

Description

Calculate how frequently each planning-unit/action assignment is selected across the stored solutions in a `solutionset-class` object.

Usage

```
selection_frequency(x)
```

Arguments

x	A `solutionset-class` object returned by `solve`.

Details

Selection frequency is calculated at the planning-unit/action level. This is the canonical decision representation used by this function because it preserves differences between solutions that select the same planning unit but assign different actions.

For each planning-unit/action pair, the frequency is:

$$f_{ia} = \frac{\sum_{s \in S} x_{ias}}{|S|},$$

where x_{ias} equals one when planning unit i receives action a in solution s, and zero otherwise.

The result is computed over all stored solutions in the supplied `SolutionSet`. To calculate frequencies for only a subset of solutions, first use `solution_filter` or `solution_unique`.

For simple conservation-planning problems without explicit actions, selected planning units are represented using the canonical action name `"conservation"`.

Selection frequency measures recurrence across the supplied solutions. It should not automatically be interpreted as formal irreplaceability because it depends on the solutions included, their sampling across objective space, and whether duplicate or dominated solutions have been retained.

Value

A `data.frame` with one row per planning-unit/action pair and the following columns:

- `pu`: planning-unit identifier;
- `action`: action identifier or name;
- `n_selected`: number of stored solutions in which the planning-unit/action pair is selected;
- `n_solutions`: total number of stored solutions considered;
- `frequency`: proportion of stored solutions in which the pair is selected.

See Also

selection_similarity, solution_filter, solution_unique, get_actions, get_pu

Examples

```
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(actions$id, each = 2),
  feature = rep(features$id, times = 2),
  multiplier = c(
    1.0, 1.0,
    1.5, 1.5
  )
)

problem <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  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"),
    runs = run_grid(
      n = 5,
      include_extremes = TRUE
    ),
    normalize_weights = TRUE
  )

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

  solutions <- solve(problem)

  # Frequency across all stored solutions
  frequency <- selection_frequency(solutions)
  frequency

  # Restrict the analysis to non-dominated solutions
  if (requireNamespace("moocore", quietly = TRUE)) {
    nondominated_solutions <- solution_filter(
      solutions,
      feasible_only = TRUE,
      nondominated = TRUE
    )

    selection_frequency(nondominated_solutions)
  }

  # Give each distinct decision configuration the same weight
  unique_solutions <- solution_unique(
    solutions,
    by = "decisions"
  )

  unique_frequency <- selection_frequency(
    unique_solutions
  )

  unique_frequency
}
```

`selection_similarity` *Calculate structural similarity among solutions*

Description

Calculate pairwise structural similarity among the stored solutions in a `solutionset-class` object.

Usage

```
selection_similarity(
  x,
  metric = c("jaccard", "hamming"),
  format = c("long", "matrix")
)
```

Arguments

`x`	A `solutionset-class` object returned by `solve`.
`metric`	Character. Similarity metric to use. One of `"jaccard"` or `"hamming"`.
`format`	Character. Output format. If `"long"`, return one row per pair of solutions. If `"matrix"`, return a symmetric similarity matrix with solution ids as row and column names.

Details

Solutions are compared using their complete planning-unit/action assignment vectors. Consequently, two solutions that select the same planning unit but assign different actions are treated as structurally different.

For simple conservation-planning problems without explicit actions, selected planning units are represented using the canonical action name `"conservation"`.

Two similarity metrics are supported:

- `"jaccard"` compares the sets of selected planning-unit/action assignments:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

  Jaccard similarity focuses on selected assignments and ignores joint absences. It is generally the preferred metric for sparse conservation and management portfolios.

- `"hamming"` calculates the proportion of decision-vector positions that are equal:

$$H(A, B) = \frac{1}{m} \sum_{k=1}^{m} I(A_k = B_k),$$

  where m is the number of feasible planning-unit/action assignments. Unlike Jaccard similarity, Hamming similarity includes shared non-selections.

For both metrics, similarity ranges from zero to one:

- `1` indicates identical assignment structures;
- `0` indicates no structural agreement under the selected metric.

The corresponding distance is calculated as:

$$D(A, B) = 1 - S(A, B).$$

The comparison is performed over all stored solutions in the supplied object. To compare only a subset, first use `solution_filter` or `solution_unique`.

Value

If `format = "long"`, a `data.frame` with columns:

- `solution_id_1`;
- `solution_id_2`;
- `similarity`;
- `distance`.

If `format = "matrix"`, a symmetric numeric matrix of similarities is returned. Its diagonal is equal to one.

The selected metric is stored in the `"metric"` attribute.

See Also

`selection_frequency`, `solution_filter`, `solution_unique`, `get_actions`, `get_pu`

Examples

```
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(actions$id, each = 2),
  feature = rep(features$id, times = 2),
  multiplier = c(
    1.0, 1.0,
```

```
    1.5, 1.5
  )
)

problem <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  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"),
    runs = run_grid(
      n = 5,
      include_extremes = TRUE
    ),
    normalize_weights = TRUE
  )

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

  solutions <- solve(problem)

  # Pairwise Jaccard similarity in long format
  jaccard_long <- selection_similarity(
    solutions
  )

  jaccard_long

  # Symmetric Jaccard similarity matrix
  jaccard_matrix <- selection_similarity(
    solutions,
    format = "matrix"
  )
```

```
  jaccard_matrix

  # Hamming similarity includes shared non-selections
  hamming_long <- selection_similarity(
    solutions,
    metric = "hamming"
  )

  hamming_long

  # Compare only structurally unique solutions
  unique_solutions <- solution_unique(
    solutions,
    by = "decisions"
  )

  selection_similarity(
    unique_solutions,
    format = "matrix"
  )
}
```

`set_method_augmecon` *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,
  runs = NULL,
  grid = NULL,
  n_points = NULL,
```

```
  include_extremes = NULL,
  lexicographic = TRUE,
  lexicographic_tol = 1e-09,
  augmentation = 0.001,
  control = NULL
)
```

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.
runs	A run design created with run_grid or run_manual. For AUGMECON, run_grid() requests automatic epsilon-level generation for secondary objectives, while run_manual() requires columns named eps_<alias> for each secondary objective.
grid	Deprecated. Previous manual-grid argument. It must be a named list with one numeric vector per secondary objective. New code should use runs = run_manual(...) instead.
n_points	Deprecated. Previous automatic-grid argument. New code should use runs = run_grid(n = ...) instead.
include_extremes	Deprecated. Previous automatic-grid argument. New code should use runs = run_grid(n = ..., include_extremes = ...) instead.
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.
control	A control object created with mo_control. It controls how infeasible runs, runs without a solution, and unexpected errors are handled. It also stores technical AUGMECON settings such as slack_upper_bound.

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 single-objective subproblem 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.

Run designs

AUGMECON runs are specified through the `runs` argument. This argument must be created with either `run_grid` or `run_manual`.

`run_grid(n = ...)` requests automatic generation of epsilon levels for the secondary objectives during `solve`. In that case, the method first computes extreme points and payoff-table ranges for the secondary objectives, and then generates `n` levels for each one.

`run_manual()` allows users to provide explicit epsilon combinations. In manual AUGMECON runs, each row is one optimization run and columns must be named `eps_<alias>`, where `<alias>` is the alias of a secondary objective. For example, if `primary = "benefit"` and `aliases = c("benefit", "cost", "loss")`, the manual run table must contain columns `eps_cost` and `eps_loss`.

In `run_manual()`, each row is used exactly as supplied. The function does not automatically create a Cartesian product of epsilon values. If a Cartesian product is desired, it should be created explicitly by the user, for example with `expand.grid`, and then passed to `run_manual()`.

The older arguments `grid`, `n_points`, and `include_extremes` are deprecated. They are still accepted for backwards compatibility and are internally converted to `run_grid()` or `run_manual()` designs.

Automatic epsilon grids

When `runs = run_grid(n = ...)` is used, the epsilon design is not built immediately. Instead, it is constructed later during `solve` using extreme-point or payoff-table information.

For each secondary objective, `run_grid()` generates a sequence of epsilon levels. With multiple secondary objectives, the final AUGMECON design is the Cartesian product of these sequences. Therefore, the number of runs can grow quickly as the number of secondary objectives increases.

If `include_extremes = TRUE` is supplied inside `run_grid()`, the automatic grid includes the extreme values of each secondary objective. Otherwise, only interior values are used.

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`.

Manual epsilon runs

Manual run designs are the most explicit way to use AUGMECON, especially when more than two objectives are involved or when only selected epsilon combinations should be explored.

For example, with one primary objective and two secondary objectives, a manual run design may contain:

```
data.frame(
  eps_cost = c(4, 6, 8),
  eps_loss = c(0, 1, 1)
)
```

This creates three runs, not a full Cartesian grid. To create all combinations, use `expand.grid()` before calling `run_manual()`.

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.

Failure handling and technical controls

The `control` argument controls how failed runs and technical AUGMECON settings are handled. It must be created with `mo_control`.

Some epsilon combinations may define infeasible subproblems. By default, failed runs can be retained in the returned `SolutionSet` with missing objective values, while feasible runs are preserved. Alternatively, users can request that the solve stops when an infeasible run, a run without a solution, or an unexpected error is encountered.

The `control` object also stores technical AUGMECON settings such as `slack_upper_bound`, the positive upper bound imposed on explicit slack variables associated with secondary objectives.

Stored configuration

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

- `name = "augmecon"`,
- `type = "augmecon"`,
- the primary objective alias,
- the full set of participating aliases,
- the set of secondary aliases,
- `runs`,
- lexicographic configuration,
- `augmentation`,
- `control`.

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

run_grid, run_manual, mo_control, 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"),
  runs = run_grid(n = 5, include_extremes = TRUE),
  lexicographic = TRUE,
  augmentation = 1e-3
)

x1$data$method

# Manual runs for one secondary objective
aug_runs <- data.frame(
  eps_cost = c(4, 6, 8)
)

x2 <- set_method_augmecon(
  x,
  primary = "benefit",
  aliases = c("benefit", "cost"),
  runs = run_manual(aug_runs),
  augmentation = 1e-3
)

x2$data$method

# Manual runs for two secondary objectives
aug_runs_3obj <- data.frame(
  eps_cost = c(4, 6, 8),
  eps_loss = c(0, 1, 1)
)

x3 <- set_method_augmecon(
  x,
  primary = "benefit",
  aliases = c("benefit", "cost", "loss"),
  runs = run_manual(aug_runs_3obj),
  augmentation = 1e-3
)

x3$data$method

# Cartesian epsilon design created explicitly by the user
aug_cartesian <- expand.grid(
  eps_cost = c(4, 6, 8),
  eps_loss = c(0, 1),
  KEEP.OUT.ATTRS = FALSE
)

x4 <- set_method_augmecon(
  x,
  primary = "benefit",
  aliases = c("benefit", "cost", "loss"),
  runs = run_manual(aug_cartesian),
  augmentation = 1e-3
```

```
)

x4$data$method

# Backwards-compatible deprecated usage
x5 <- 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
)

x5$data$method

# Control failure handling and the AUGMECON slack upper bound
x6 <- set_method_augmecon(
  x,
  primary = "benefit",
  aliases = c("benefit", "cost"),
  runs = run_manual(data.frame(eps_cost = c(4, 6, 8))),
  augmentation = 1e-3,
  control = mo_control(
    stop_on_infeasible = FALSE,
    stop_on_no_solution = FALSE,
    stop_on_error = TRUE,
    slack_upper_bound = 1e6
  )
)

x6$data$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. Multiple subproblems are generated using a run design supplied through `runs`.

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,
  aliases = NULL,
  runs = NULL,
  eps = NULL,
  mode = NULL,
  n_points = NULL,
  include_extremes = NULL,
  lexicographic = TRUE,
  lexicographic_tol = 1e-08,
  control = NULL
)
```

Arguments

x	A Problem object.
primary	Character string giving the alias of the primary objective to optimize directly.
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.
runs	A run design created with run_grid or run_manual. For epsilon-constraint methods, run_grid() requests automatic epsilon-level generation, while run_manual() requires columns named eps_<alias> for each constrained objective.
eps	Deprecated. Epsilon specification used by the previous mode = "manual" interface. It may be a named numeric vector or a named list of numeric vectors. New code should use runs = run_manual(...) instead.
mode	Deprecated. Previous interface selector, either "manual" or "auto". New code should use runs = run_manual(...) or runs = run_grid(...) instead.
n_points	Deprecated. Previous automatic-grid argument. New code should use runs = run_grid(n = ...) instead.
include_extremes	Deprecated. Previous automatic-grid argument. New code should use runs = run_grid(n = ..., include_extremes = ...) instead.
lexicographic	Logical scalar. If TRUE, compute automatic-grid extreme points lexicographically when runs = run_grid(...) is used.
lexicographic_tol	Numeric scalar ≥ 0. Tolerance used in lexicographic extreme-point computation.
control	A control object created with mo_control. It controls how infeasible runs, runs without a solution, and unexpected errors are handled.

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 the sense of each objective, the internal implementation may transform minimization and maximization objectives into equivalent solver-ready constrained forms. 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 trade-off solutions.

Run designs

Epsilon-constraint runs are specified through the `runs` argument. This argument must be created with either `run_grid` or `run_manual`.

`run_grid(n = ...)` requests automatic generation of epsilon levels during `solve`. The epsilon levels are computed from extreme-point or payoff information. In the current implementation, `run_grid()` for epsilon-constraint supports exactly two objectives: one primary objective and one constrained objective.

`run_manual()` allows users to provide explicit epsilon combinations. In manual epsilon-constraint runs, each row is one optimization run and columns must be named `eps_<alias>`, where `<alias>` is the alias of a constrained objective. For example, if `primary = "benefit"` and `aliases = c("benefit", "cost", "loss")`, the manual run table must contain columns `eps_cost` and `eps_loss`.

In `run_manual()`, each row is used exactly as supplied. The function does not automatically create a Cartesian product of epsilon values. If a Cartesian product is desired, it should be created explicitly by the user, for example with `expand.grid`, and then passed to `run_manual()`.

The older arguments `eps`, `mode`, `n_points`, and `include_extremes` are deprecated. They are still accepted for backwards compatibility and are internally converted to `run_grid()` or `run_manual()` designs.

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.

Automatic epsilon grids

When `runs = run_grid(n = ...)` is used, the epsilon grid is not built immediately. Instead, it is constructed later during `solve` using extreme-point or payoff information.

In the current implementation, automatic epsilon-grid generation supports exactly two objectives:

- one primary objective,
- one constrained objective.

Therefore, if `runs = run_grid(...)`, then `aliases` must contain exactly two objective aliases. Problems with three or more objectives must use `runs = run_manual(...)`.

If `include_extremes = TRUE` is supplied inside `run_grid()`, 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.

Manual epsilon runs

Manual run designs support two or more objectives. They are the general way to use the epsilon-constraint method when more than two objectives are involved.

For example, with one primary objective and two constrained objectives, a manual run design may contain:

```
data.frame(
  eps_cost = c(4, 6, 8),
  eps_loss = c(0, 1, 1)
)
```

This creates three runs, not a full Cartesian grid. To create all combinations, use `expand.grid()` before calling `run_manual()`.

Failure handling

The `control` argument controls how failed runs are handled. It must be created with `mo_control`.

Some epsilon levels may define infeasible subproblems. By default, failed runs can be retained in the returned `SolutionSet` with missing objective values, while feasible runs are preserved. Alternatively, users can request that the solve stops when an infeasible run, a run without a solution, or an unexpected error is encountered.

Stored configuration

The configured method stores:

- `name` = `"epsilon_constraint"`,
- `type` = `"epsilon_constraint"`,
- `primary`,
- `aliases`,
- `constrained`,
- `runs`,
- lexicographic configuration,
- `control`.

With `runs = run_grid(...)`, the actual epsilon design is generated later during `solve`. With `runs = run_manual(...)`, the explicit user-supplied run design is stored and then used by `solve`.

Value

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

See Also

`run_grid`, `run_manual`, `mo_control`, `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")

# Automatic epsilon grid for two objectives
x1 <- set_method_epsilon_constraint(
  x,
  primary = "benefit",
  aliases = c("benefit", "cost"),
  runs = run_grid(n = 5, include_extremes = TRUE),
  lexicographic = TRUE,
  lexicographic_tol = 1e-8
)

x1$data$method

# Manual runs with one constrained objective
eps_runs <- data.frame(
  eps_cost = c(4, 6, 8)
)

x2 <- set_method_epsilon_constraint(
  x,
  primary = "benefit",
  aliases = c("benefit", "cost"),
  runs = run_manual(eps_runs)
)

x2$data$method

# Manual runs with more than two objectives
eps_runs_3obj <- data.frame(
  eps_cost = c(4, 6, 8),
  eps_loss = c(0, 1, 1)
```

```
)

x3 <- set_method_epsilon_constraint(
  x,
  primary = "benefit",
  aliases = c("benefit", "cost", "loss"),
  runs = run_manual(eps_runs_3obj)
)

x3$data$method

# Cartesian epsilon design created explicitly by the user
eps_cartesian <- expand.grid(
  eps_cost = c(4, 6, 8),
  eps_loss = c(0, 1),
  KEEP.OUT.ATTRS = FALSE
)

x4 <- set_method_epsilon_constraint(
  x,
  primary = "benefit",
  aliases = c("benefit", "cost", "loss"),
  runs = run_manual(eps_cartesian)
)

x4$data$method

# Backwards-compatible deprecated usage
x5 <- set_method_epsilon_constraint(
  x,
  primary = "benefit",
  aliases = c("benefit", "cost"),
  mode = "manual",
  eps = list(cost = c(4, 6, 8))
)

x5$data$method

# Control failure handling
x6 <- set_method_epsilon_constraint(
  x,
  primary = "benefit",
  aliases = c("benefit", "cost"),
  runs = run_manual(data.frame(eps_cost = c(4, 6, 8))),
  control = mo_control(
    stop_on_infeasible = FALSE,
    stop_on_no_solution = FALSE,
    stop_on_error = TRUE
  )
)

x6$data$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,
  runs = NULL,
  weights = NULL,
  normalize_weights = TRUE,
  objective_scaling = FALSE,
  control = NULL
)
```

Arguments

x	A Problem object.
aliases	Character vector of objective aliases to combine. Each alias must correspond to a previously registered atomic objective.
runs	A run design created with run_grid or run_manual. For weighted-sum methods, automatic grids define weight combinations, while manual runs must contain columns named weight_<alias>.
weights	Deprecated. Numeric vector of weights, with the same length and order as aliases. This argument is kept for backwards compatibility and is internally converted to runs = run_manual(...). New code should use runs instead.
normalize_weights	Logical. If TRUE, normalize the weights in each run to sum to one before solving.
objective_scaling	Logical. If TRUE, request scaling of the participating objectives before weighted aggregation in the solving layer.
control	A control object created with mo_control. It controls how infeasible runs, runs without a solution, and unexpected errors are handled.

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 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 responsible for constructing a solver-ready scalar objective from the stored objective specifications and the requested weights.

Run designs

Weighted-sum runs are specified through the `runs` argument. This argument must be created with either `run_grid` or `run_manual`.

`run_grid(n = ...)` automatically generates a grid of weight combinations. For two objectives, this is a regular sequence of weights along the line between the two objectives. For three or more objectives, the grid is generated over the weight simplex, where all weights are non-negative and sum to one.

`run_manual()` allows users to provide explicit weight combinations. In manual weighted-sum runs, each row is one optimization run and columns must be named `weight_<alias>`. For example, if `aliases = c("cost", "benefit")`, the manual run table must contain columns `weight_cost` and `weight_benefit`.

The older `weights` argument is deprecated. It is still accepted for backwards compatibility and is internally converted to a one-row `run_manual()` design.

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 weights in each run are rescaled to sum to one:

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

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`, each row of weights must already sum to one.

Objective scaling

If `objective_scaling = TRUE`, the solving layer scales 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 in the solving layer.

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 standardizes them internally before building the scalar objective.

Users should provide non-negative weights according to the original meaning of each objective. For example, a positive weight on a maximization objective means that higher values of that objective are preferred.

Failure handling

The `control` argument controls how failed runs are handled. It must be created with `mo_control`.

Weighted-sum runs do not normally introduce additional constraints, so they should not usually create infeasible subproblems by themselves. However, runs may still fail if the underlying model is infeasible, the solver stops before finding a feasible solution, or a numerical/modeling issue occurs. The `control` argument determines whether such failures stop the whole solve or are retained in the returned `SolutionSet` with missing objective values.

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",
- type = "weighted",
- aliases,
- runs,
- normalize_weights,
- objective_scaling,
- control.

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

run_grid, run_manual, mo_control, 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")

# Automatic weight grid
x1 <- set_method_weighted_sum(
  x,
  aliases = c("cost", "benefit"),
  runs = run_grid(n = 5, include_extremes = TRUE),
  objective_scaling = TRUE
)

x1$data$method

# Manual weighted runs
manual_weights <- data.frame(
  weight_cost = c(1.0, 0.75, 0.50, 0.25, 0.0),
  weight_benefit = c(0.0, 0.25, 0.50, 0.75, 1.0)
)

x2 <- set_method_weighted_sum(
  x,
  aliases = c("cost", "benefit"),
  runs = run_manual(manual_weights),
  normalize_weights = FALSE,
  objective_scaling = TRUE
)

x2$data$method

# Manual runs with automatic weight normalization
manual_weights2 <- data.frame(
  weight_cost = c(2, 1, 1),
  weight_benefit = c(1, 1, 3)
)

x3 <- set_method_weighted_sum(
  x,
  aliases = c("cost", "benefit"),
  runs = run_manual(manual_weights2),
  normalize_weights = TRUE
)

x3$data$method
```

```
# Backwards-compatible deprecated usage
x4 <- set_method_weighted_sum(
  x,
  aliases = c("cost", "benefit"),
  weights = c(0.4, 0.6),
  normalize_weights = FALSE
)

x4$data$method

# Control failure handling
x5 <- set_method_weighted_sum(
  x,
  aliases = c("cost", "benefit"),
  runs = run_grid(n = 5),
  control = mo_control(
    stop_on_infeasible = TRUE,
    stop_on_no_solution = TRUE,
    stop_on_error = TRUE
  )
)

x5$data$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_append` *Append solutions from another solution set*

Description

Append the runs and stored solutions from one `solutionset-class` object to another.

This function combines two `SolutionSet` objects generated from the same planning problem. It is intended for combining results obtained with different **multiscape** solving workflows, such as weighted-sum, epsilon-constraint, or AUGMECON methods, while keeping a single coherent result object for downstream extraction, plotting, and analysis.

Usage

```
solution_append(x, y)
```

Arguments

`x`	A `solutionset-class` object returned by `solve`.
`y`	A second `solutionset-class` object returned by `solve` and generated from the same planning problem as `x`.

Details

`solution_append()` is a solution-set management function. It does not modify the original input objects. Instead, it returns a new `SolutionSet` containing the runs and solutions from both inputs.

Both input objects must be generated from the same planning problem. This is checked conservatively before appending. In particular, the two objects must have compatible:

- planning units;
- features and feature distributions;
- actions, feasible action pairs, and effects;
- profit data, when present;
- targets;
- locks and constraints;
- spatial relations;
- objective specifications.

Differences in method settings, run design, solver settings, solver status, runtime, gaps, and other solve diagnostics are allowed.

The appended runs and solutions are assigned new `run_id` and `solution_id` values to keep identifiers unique in the combined object. Identifiers are not required to match between the two inputs.

This function is not intended to combine results from different planning problems, scenarios, target sets, or objective definitions. Such workflows should be handled by a future comparison/binding function rather than by `solution_append()`.

Value

A new solutionset-class object containing the runs and stored solutions from both input objects.

See Also

solution_filter, solutionset-class, get_runs, get_objectives, plot_tradeoff

Examples

```
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(actions$id, each = 2),
  feature = rep(features$id, times = 2),
  multiplier = c(
    1.0, 1.0,
    1.5, 1.5
  )
)

make_problem <- function() {
  create_problem(
    pu = pu,
    features = features,
    dist_features = dist_features,
    cost = "cost"
  ) |>
    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")
}

weighted_problem <- make_problem() |>
  set_method_weighted_sum(
    aliases = c("cost", "benefit"),
    runs = run_grid(
      n = 4,
      include_extremes = TRUE
    ),
    normalize_weights = TRUE
  )

epsilon_problem <- make_problem() |>
  set_method_epsilon_constraint(
    primary = "cost",
    runs = run_grid(
      n = 4,
      include_extremes = TRUE
    )
  )

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

  epsilon_problem <- set_solver_cbc(
    epsilon_problem,
    verbose = FALSE
  )

  weighted_solutions <- solve(weighted_problem)
  epsilon_solutions <- solve(epsilon_problem)

  combined_solutions <- solution_append(
    weighted_solutions,
    epsilon_solutions
  )

  # Inspect the combined run history
  get_runs(combined_solutions)

  # Inspect objective values from both methods
  get_objectives(
    combined_solutions,
```

```
    format = "wide"
  )

  # The original objects remain unchanged
  get_runs(weighted_solutions)
  get_runs(epsilon_solutions)
}
```

`solution_filter` *Filter solutions in a solution set*

Description

Return a reduced `solutionset-class` object containing only the runs or solutions that match the requested filters.

This function is intended to curate `SolutionSet` objects before downstream analysis, plotting, frontier analysis, or post-hoc evaluation. It filters all relevant components of the object consistently, including the run table, design table, stored run-level solutions, and available summary tables.

Usage

```
solution_filter(
  x,
  run_id = NULL,
  solution_id = NULL,
  status = NULL,
  feasible_only = FALSE,
  nondominated = FALSE,
  objectives = NULL
)
```

Arguments

`x`	A `solutionset-class` object returned by `solve`.
`run_id`	Optional integer vector of run ids to keep.
`solution_id`	Optional character vector of solution ids to keep. Runs without a stored solution are never matched by this filter.
`status`	Optional character vector of run statuses to keep. Matching is case-insensitive.
`feasible_only`	Logical. If `TRUE`, keep only runs whose status is interpreted as having produced a usable solution. The current accepted statuses are `"optimal"`, `"feasible"`, `"suboptimal"`, `"time_limit"`, and `"gap_limit"`.
`nondominated`	Logical. If `TRUE`, keep only non-dominated solutions among the runs retained by the other filters. This uses **moocore** internally.
`objectives`	Optional character vector of objective names to use when `nondominated = TRUE`. If `NULL`, all available objective-value columns are used.

Details

A `SolutionSet` distinguishes between runs and stored solutions.

- `run_id` identifies a run or attempted solve. Runs may be feasible, optimal, infeasible, failed, or otherwise incomplete.
- `solution_id` identifies a stored solution. Runs that did not produce a solution have `solution_id = NA`.

Therefore, filtering by `run_id` and filtering by `solution_id` are not always equivalent. For example, an infeasible run may have a `run_id` but no `solution_id`.

The function filters:

- `x$solution$runs`, using the selected `run_id`s;
- `x$solution$design`, when it contains a `run_id` column;
- `x$solution$solutions`, using the selected `solution_id`s;
- all tables in `x$summary` that contain a `run_id` column.

The function does not renumber `run_id` or `solution_id`. This preserves traceability to the original run design.

If more than one filter is supplied, filters are combined using logical *and*. For example, setting both `status = "optimal"` and `solution_id = c("s1", "s3")` keeps only optimal runs whose `solution_id` is either `"s1"` or `"s3"`.

If `nondominated = TRUE`, the function further keeps only non-dominated solutions among the runs retained by the previous filters. Dominance is evaluated in objective space using the objective values stored in the run table. Objective senses are read from `get_objective_specs`; any maximization objective is internally multiplied by -1 so that dominance can be evaluated in minimization space.

Non-dominated filtering requires the **moocore** package.

Value

A filtered `solutionset-class` object.

See Also

`solutionset-class`, `solve`, `get_runs`, `get_objectives`, `get_objective_specs`, `get_pu`, `get_actions`

Examples

```
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(actions$id, each = 2),
  feature = rep(features$id, times = 2),
  multiplier = c(
    1.0, 1.0,
    1.5, 1.5
  )
)

problem <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  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"),
    runs = run_grid(
      n = 5,
      include_extremes = TRUE
    ),
    normalize_weights = TRUE
  )

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

```
  solutions <- solve(problem)
  runs <- get_runs(solutions)

  # Keep only runs with a usable solver status
  feasible_solutions <- solution_filter(
    solutions,
    feasible_only = TRUE
  )

  # Keep selected runs
  selected_runs <- solution_filter(
    solutions,
    run_id = runs$run_id[1:2]
  )

  # Keep one stored solution
  solution_ids <- runs$solution_id[
    !is.na(runs$solution_id)
  ]

  if (length(solution_ids) > 0L) {
    selected_solution <- solution_filter(
      solutions,
      solution_id = solution_ids[1]
    )
  }

  # Keep only optimal runs
  if ("optimal" %in% tolower(runs$status)) {
    optimal_solutions <- solution_filter(
      solutions,
      status = "optimal"
    )
  }

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

    # Evaluate dominance using selected objectives
    nondominated_subset <- solution_filter(
      solutions,
      feasible_only = TRUE,
      nondominated = TRUE,
      objectives = c("cost", "benefit")
    )
  }
}
```

`solution_unique` *Keep unique solutions in a solution set*

Description

Return a reduced `solutionset-class` object containing one representative from each group of equivalent solutions.

Solutions can be considered equivalent according to either their decision vectors or their objective values.

Usage

```
solution_unique(
  x,
  by = c("decisions", "objectives"),
  keep = c("first", "last"),
  objectives = NULL,
  tolerance = sqrt(.Machine$double.eps)
)
```

Arguments

`x`	A `solutionset-class` object returned by `solve`.
`by`	Character. Definition of uniqueness. One of `"decisions"` or `"objectives"`.
`keep`	Character. Which representative to retain from each group of equivalent solutions. If `"first"`, retain the first solution in the current run order. If `"last"`, retain the last.
`objectives`	Optional character vector of objective names to compare when `by = "objectives"`. If `NULL`, all available objectives are used. This argument is ignored when `by = "decisions"`.
`tolerance`	Non-negative numeric tolerance used when comparing objective values. It is only used when `by = "objectives"`.

Details

`solution_unique()` is a solution-set management function. It removes repeated solutions while consistently filtering the run table, design table, stored run-level solutions, and summary tables.

Two definitions of uniqueness are supported:

- `by = "decisions"` compares the complete stored decision vector of each solution. Two solutions are considered equivalent when their decision vectors are identical. This identifies repeated planning-unit or planning-unit/action configurations, even when they were generated by different runs or multi-objective parameter combinations.

- by = "objectives" compares the selected objective values. Two solutions are considered equivalent when all compared objective values are equal within the specified numerical tolerance. Such solutions may still have different decision vectors.

Consequently, uniqueness in objective space and uniqueness in decision space are not equivalent. Two spatially different solutions may produce the same objective values, while repeated runs may generate exactly the same decision vector.

Only runs with a stored solution_id can be assessed. Runs without a stored solution, such as infeasible runs, are preserved unchanged. This retains the full run history while removing only duplicated stored solutions.

The function does not renumber run_id or solution_id. The representative retained from each duplicate group keeps its original identifiers, preserving traceability to the original run design.

For by = "objectives", numerical equality is assessed using a relative comparison:

$$|a - b| \leq \epsilon \max(1, |a|, |b|),$$

where ϵ is specified by tolerance. This avoids treating insignificant floating-point differences as distinct objective points.

Value

A new solutionset-class object containing one representative from each group of equivalent stored solutions, together with any runs that did not produce a stored solution.

See Also

solution_filter, solution_append, get_objectives, get_solution_vector

Examples

```
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(actions$id, each = 2),
  feature = rep(features$id, times = 2),
  multiplier = c(
    1.0, 1.0,
    1.5, 1.5
  )
)

problem <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  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"),
    runs = run_grid(
      n = 7,
      include_extremes = TRUE
    ),
    normalize_weights = TRUE
  )

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

  solutions <- solve(problem)

  # Keep one representative for each distinct decision vector
  unique_decisions <- solution_unique(
    solutions,
    by = "decisions"
  )

  # Keep one representative for each distinct objective point
```

```
  unique_objectives <- solution_unique(
    solutions,
    by = "objectives"
  )

  # Compare only selected objectives
  unique_cost_benefit <- solution_unique(
    solutions,
    by = "objectives",
    objectives = c("cost", "benefit")
  )

  # Keep the last representative of each duplicate group
  unique_last <- solution_unique(
    solutions,
    by = "decisions",
    keep = "last"
  )

  # Compare the number of stored solutions
  sum(!is.na(get_runs(solutions)$solution_id))
  sum(!is.na(get_runs(unique_decisions)$solution_id))
  sum(!is.na(get_runs(unique_objectives)$solution_id))

  # Typical cleaning workflow
  if (requireNamespace("moocore", quietly = TRUE)) {
    clean_solutions <- solutions |>
      solution_filter(
        feasible_only = TRUE,
        nondominated = TRUE
      ) |>
      solution_unique(
        by = "decisions"
      )
  }
}
```

`solutionset-class` *SolutionSet class*

Description

The `SolutionSet` class stores the result of solving a `Problem` object.

A `SolutionSet` object is the standard result object returned by `solve`. It may contain a single solved run, as in ordinary single-objective optimization, or multiple solved runs, as in weighted-sum, ϵ-constraint, or AUGMECON workflows.

The object stores the original problem, run-level design and outcome tables, objective values, diagnostics, method metadata, and additional metadata needed to inspect, compare, and analyse optimization results.

Details

Conceptual role

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

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

Single-objective workflows are represented as `SolutionSet` objects with one run. Multi-objective workflows are represented as `SolutionSet` objects with one or more runs generated by the selected method.

This design provides a common result structure for downstream analysis, including objective-value extraction, run diagnostics, trade-off analysis, Pareto-frontier summaries, distance-based ranking, and comparison between scenarios.

Typical use cases

A `SolutionSet` is used to inspect and analyse the results returned by `solve`. Typical use cases include:

- inspecting selected planning units or actions;
- extracting run-level objective values;
- comparing alternative solutions produced by a multi-objective method;
- analysing trade-offs between objectives;
- filtering feasible, unique, or non-dominated solutions;
- computing distances to observed ideal or nadir points;
- preparing plots and tables for reporting.

Internal structure

A `SolutionSet` object separates information into several layers:

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

`solution` A named `list` containing the core solving outputs. This typically includes the run design, the run summary table, and the stored run-level solution objects used internally by **multiscape**.

`summary` A named `list` containing user-facing summaries aggregated across runs when such summaries are available.

`diagnostics` A named `list` containing diagnostics about the solution set as a whole.

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

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

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

Run-level content

The `solution` field is the main internal container for run-level information. Typical entries include:

`design` A `data.frame` describing the optimization design. In multi-objective workflows this may include weights, ϵ-levels, or other method-specific parameters. In single-objective workflows this table usually contains one row.

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

`solutions` A list of stored run-level solution objects used internally by **multiscape**. Users should normally rely on accessor functions rather than directly manipulating this list.

In most workflows, the `runs` table is the most useful compact representation of the solution set. Detailed inspection should preferably be done through accessor functions such as `get_pu`, `get_actions`, `get_features`, and `get_targets`.

Objective values and run tables

The run table may store objective values in columns named `value_<alias>`, where `<alias>` is the alias of a registered objective. For example:

- `value_cost`,
- `value_frag`,
- `value_benefit`.

This convention provides a compact format for downstream functions that analyse trade-offs, frontiers, or distances between solutions.

Depending on the solving method, the run table may also contain design columns such as:

- `weight_<alias>` for weighted-sum runs;
- `eps_<alias>` for ϵ-constraint or AUGMECON runs.

Single-objective and multi-objective results

`solve()` always returns a `SolutionSet`. For a single-objective problem, the returned object contains one run. For a multi-objective workflow, the returned object contains one or more runs generated by the selected method.

This unified output structure makes it possible to use the same inspection and analysis functions regardless of whether the original problem was single-objective or multi-objective.

Diagnostics

The `diagnostics` field stores metadata about the solve process. Depending on the workflow, it may summarize:

- number of design rows;
- number of attempted runs;
- number of stored solutions;
- status frequencies;
- runtime ranges;
- gap ranges;
- and other aggregate information about the set of runs.

Printing

The `print()` method provides a concise summary of the solution set. It reports:

- the optimization method and participating objective aliases;

- the run-design type when this information is available;
- the number of design rows, attempted runs, stored solutions, and runs without a stored solution;
- run-level status, runtime, and optimality-gap summaries;
- the observed range of each objective across stored solutions;
- and any filtering, uniqueness, or append metadata already recorded in the object.

Printing does not calculate non-dominance, uniqueness, similarities, selection frequencies, or distances. These analyses are intentionally kept outside the print method so that printing remains fast and does not depend on optional packages.

This printed output is intended as a quick overview. Detailed inspection should use public accessor and analysis functions such as `get_runs`, `get_objectives`, `solution_filter`, and `frontier_distances`.

Value

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

Fields

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

`solution` A named `list` containing the core solving outputs, typically including `design`, `runs`, and internally stored run-level solutions.

`summary` A named `list` containing user-facing summaries associated with the solution set.

`diagnostics` A named `list` containing diagnostics about the solution set as a whole.

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

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

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

Methods

`print()` Print a concise summary of the solution set, including method name, number of runs, and run-level diagnostics.

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

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

`getMethod()` Return the method specification stored in `self$method`.

`getDesign()` Return the design table stored in `self$solution$design`.

`getRuns()` Return the run summary table stored in `self$solution$runs`.

`getSolutions()` Return the internally stored run-level solution objects. This is mainly intended for advanced use; most users should prefer higher-level accessor functions.

See Also

`solve`, `plot_tradeoff`, `get_pu`, `get_actions`, `get_features`, `get_targets`

solve *Solve a planning problem*

Description

Solve a planning problem stored in a Problem object.

This is the main execution step of the **multiscape** workflow. It reads the problem specification stored in a Problem object, builds the corresponding optimization model when needed, applies the configured solver settings, and returns a solutionset-class object.

A SolutionSet is the standard result object returned by solve(). Single-objective workflows are represented as one-run SolutionSet objects, while multi-objective workflows are represented as multi-run SolutionSet objects.

Usage

```
solve(x, ...)

## S3 method for class 'Problem'
solve(x, ...)
```

Arguments

x	A Problem object created with create_problem and optionally enriched with actions, effects, targets, constraints, objectives, spatial relations, method settings, and solver settings.
...	Additional arguments reserved for internal or legacy solver handling. These are not part of the main recommended user interface.

Details

Role of solve()

The typical **multiscape** workflow is:

```
x <- create_problem(...)
x <- add_...(x, ...)
x <- set_...(x, ...)
res <- solve(x)
```

Thus, solve() is the stage at which the stored problem specification is turned into one or more optimization runs.

For most users, solve() is the standard execution entry point. Explicit compilation with compile_model is optional and is mainly useful for advanced inspection or debugging workflows.

What solve() **uses from the problem object**

The function uses the information already stored in the Problem object, including:

- baseline planning data;

- actions, effects, profit, and spatial relations;
- targets and constraints;
- registered objectives;
- an optional multi-objective method configuration;
- solver settings.

If a model has not yet been built, it is built internally during the solve process. If a model snapshot or pointer already exists, the solving layer may reuse or refresh it depending on the internal model state.

Single-objective and multi-objective behaviour

`solve()` always returns a `solutionset-class` object.

- **Single-objective case.** If exactly one objective is active and no multi-objective method is configured, `solve()` runs a single optimization problem and returns a one-run `SolutionSet`.
- **Multi-objective case.** If a multi-objective method is configured, `solve()` dispatches internally according to the stored method name and returns a multi-run `SolutionSet`.

This unified output structure makes it possible to use the same inspection, plotting, and analysis functions regardless of whether the original problem was single-objective or multi-objective.

Currently supported multi-objective method names are:

- `"weighted"`;
- `"epsilon_constraint"`;
- `"augmecon"`.

Consistency rule

If multiple objectives are registered but no multi-objective method has been selected, `solve()` stops with an error. In practical terms:

- one objective and no multi-objective method ⇒ single-objective solve;
- multiple objectives and a valid multi-objective method ⇒ multi-objective solve;
- multiple objectives and no multi-objective method ⇒ error.

Implicit conservation-planning model

If no explicit actions and no explicit effects are provided, `solve()` can build the corresponding classical conservation-planning formulation internally. In this case, selecting a planning unit is interpreted as applying an implicit conservation action, and the baseline feature amounts stored in `dist_features` count toward representation targets.

This allows standard reserve-selection problems to be solved without requiring users to manually define actions and effects. Explicit action-based workflows remain available by using `add_actions` and `add_effects`.

Solver settings

Solver configuration is read from the `Problem` object, typically after calling `set_solver` or one of its convenience wrappers such as `set_solver_gurobi` or `set_solver_cbc`.

These settings may include:

- the selected backend;
- time limits;
- optimality-gap settings;
- CPU cores;
- verbosity options;
- backend-specific solver parameters.

Method dispatch

`solve()` is an S3 generic. The public method documented here is `solve.Problem()`, which operates on `Problem` objects.

Value

A `solutionset-class` object.

The returned object contains run-level information, solver diagnostics, objective values, and stored optimization outputs. For single-objective problems, the returned `SolutionSet` contains one run. For multi-objective workflows, it contains one or more runs generated by the selected method.

Users will typically inspect or visualize results using accessor and plotting functions such as `get_pu`, `get_actions`, `get_features`, `get_targets`, and `plot_tradeoff`.

See Also

`problem-class`, `solutionset-class`, `compile_model`, `set_solver`, `set_solver_cbc`, `set_solver_gurobi`, `set_method_weighted_sum`, `set_method_epsilon_constraint`, `set_method_augmecon`, `add_actions`, `add_effects`

Examples

```
# ------------------------------------------------------------
# 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)
)

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

```
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  add_constraint_targets_relative(0.05) |>
  add_objective_min_cost(alias = "cost")

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

# ------------------------------------------------------------
# Minimal action-based example
# ------------------------------------------------------------
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(1.00, 1.00, 1.50, 1.50)
)

x_actions <- create_problem(
  pu = pu,
  features = features,
  dist_features = dist_features,
  cost = "cost"
) |>
  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_actions <- set_solver_cbc(x_actions, verbose = FALSE)
  solset_actions <- solve(x_actions)
  print(solset_actions)
}
```

Index