Package 'couplr'

March 3, 2026

Title Optimal Pairing and Matching via Linear Assignment

Version 1.1.0

Description Solves optimal pairing and matching problems using linear assignment
algorithms. Provides implementations of the Hungarian method (Kuhn 1955)
<doi:10.1002/nav.3800020109>, Jonker-Volgenant shortest path algorithm
(Jonker and Volgenant 1987) <doi:10.1007/BF02278710>, Auction algorithm
(Bertsekas 1988) <doi:10.1007/BF02186476>, cost-scaling
(Goldberg and Kennedy 1995) <doi:10.1007/BF01585996>, scaling algorithms
(Gabow and Tarjan 1989) <doi:10.1137/0218069>, push-relabel (Goldberg and
Tarjan 1988) <doi:10.1145/48014.61051>, and Sinkhorn entropy-regularized
transport (Cuturi 2013) <doi:10.48550/arxiv.1306.0895>. Designed for
matching plots, sites, samples, or any pairwise optimization problem.
Supports rectangular matrices, forbidden assignments, data frame inputs,
batch solving, k-best solutions, and pixel-level image morphing for
visualization. Includes automatic preprocessing with variable health
checks, multiple scaling methods (standardized, range, robust), greedy
matching algorithms, and comprehensive balance diagnostics for assessing
match quality using standardized differences and distribution comparisons.

License MIT + file LICENSE

Language en-US

Encoding UTF-8

RoxygenNote 7.3.3

Depends R (>= 4.1.0)

Imports Rcpp (>= 1.0.0), tibble (>= 3.0.0), dplyr (>= 1.0.0), rlang
(>= 0.4.0), purrr (>= 0.3.0), methods

Suggests testthat (>= 3.0.0), xml2, e1071, R.utils, microbenchmark,
withr, knitr, rmarkdown, bench, parallel, future (>= 1.20.0),
future.apply (>= 1.8.0), ggplot2, ggraph, tidygraph, magick,
OpenImageR, farver, av, reticulate, png, combinat

LinkingTo Rcpp, RcppEigen, testthat

SystemRequirements C++17

LazyData true

VignetteBuilder knitr

URL <https://gillescolling.com/couplr/>,
	<https://github.com/gcol33/couplr>

BugReports <https://github.com/gcol33/couplr/issues>

Config/testthat/edition 3

Config/testthat/parallel true

NeedsCompilation yes

Author Gilles Colling [aut, cre, cph] (ORCID:
	<<https://orcid.org/0000-0003-3070-6066>>)

Maintainer Gilles Colling <gilles.colling051@gmail.com>

Repository CRAN

Date/Publication 2026-03-03 11:30:02 UTC

Contents

---

assignment   *Linear assignment solver*

Description

Solve the linear assignment problem (minimum- or maximum-cost matching) using several algorithms. Forbidden edges can be marked as `NA` or `Inf`.

Usage

```
assignment(
  cost,
  maximize = FALSE,
 method = c("auto", "jv", "hungarian", "auction", "auction_gs", "auction_scaled", "sap",
   "ssp", "csflow", "hk01", "bruteforce", "ssap_bucket", "cycle_cancel", "gabow_tarjan",
   "lapmod", "csa", "ramshaw_tarjan", "push_relabel", "orlin", "network_simplex"),
  auction_eps = NULL,
```

```
    eps = NULL
)
```

Arguments

cost	Numeric matrix; rows = tasks, columns = agents. NA or Inf entries are treated as forbidden assignments.
maximize	Logical; if TRUE, maximizes the total cost instead of minimizing.
method	Character string indicating the algorithm to use. Options:

General-purpose solvers:

- "auto" — Automatic selection based on problem characteristics (default)
- "jv" — 'Jonker-Volgenant', fast general-purpose O(n³)
- "hungarian" — Classic 'Hungarian' algorithm O(n³)

Auction-based solvers:

- "auction" — 'Bertsekas' auction with adaptive epsilon
- "auction_gs" — 'Gauss-Seidel' variant, good for spatial structure
- "auction_scaled" — 'Epsilon-scaling', fastest for large dense problems

Specialized solvers:

- "sap" / "ssp" — Shortest augmenting path, handles sparsity well
- "lapmod" — Sparse JV variant, faster when >50\
- "hk01" — 'Hopcroft-Karp' for binary (0/1) costs only
- "ssap_bucket" — 'Dial' algorithm for integer costs
- "line_metric" — O(n log n) for 1D assignment problems
- "bruteforce" — Exact enumeration for tiny problems (n <= 8)

Advanced solvers:

- "csa" — 'Goldberg-Kennedy' cost-scaling, often fastest for medium-large
- "gabow_tarjan" — 'Gabow-Tarjan' bit-scaling with complementary slackness O(n³ log C)
- "cycle_cancel" — Cycle-canceling with 'Karp' algorithm
- "csflow" — Cost-scaling network flow
- "network_simplex" — 'Network simplex' with spanning tree representation
- "orlin" — 'Orlin-Ahuja' scaling O(sqrt(n) * m * log(nC))
- "push_relabel" — 'Push-relabel' max-flow based solver
- "ramshaw_tarjan" — 'Ramshaw-Tarjan', optimized for rectangular matrices (n != m)

auction_eps	Optional numeric epsilon for the 'Auction'/'Auction-GS' methods. If NULL, an internal default (e.g., 1e-9) is used.
eps	Deprecated. Use auction_eps. If provided and auction_eps is NULL, its value is used for auction_eps.

Details

method = "auto" selects an algorithm based on problem size/shape and data characteristics:

- Very small (n <= 8): "bruteforce" — exact enumeration

- Binary/constant costs: "hk01" — specialized for 0/1 costs

- Large sparse (n>100, >50\

- Sparse or very rectangular: "sap" — handles sparsity well

- Small-medium (8 < n <= 50): "hungarian" — provides exact dual solutions

- Medium (50 < n <= 75): "jv" — fast general-purpose solver

- Large (n>75): "auction_scaled" — fastest for large dense problems

Benchmarks show 'Auction-scaled' and 'JV' are 100-1500x faster than 'Hungarian' at n=500.

Value

An object of class lap_solve_result, a list with elements:

- match — integer vector of length min(nrow(cost), ncol(cost)) giving the assigned column for each row (0 if unassigned).

- total_cost — numeric scalar, the objective value.

- status — character scalar, e.g. "optimal".

- method_used — character scalar, the algorithm actually used.

See Also

- lap_solve() — Tidy interface returning tibbles

- lap_solve_kbest() — Find k-best assignments ('Murty' algorithm)

- assignment_duals() — Extract dual variables for sensitivity analysis

- bottleneck_assignment() — Minimize maximum edge cost (minimax)

- sinkhorn() — Entropy-regularized optimal transport

Examples

```
cost <- matrix(c(4,2,5, 3,3,6, 7,5,4), nrow = 3, byrow = TRUE)
res  <- assignment(cost)
res$match; res$total_cost
```

---

assignment_duals   *Solve assignment problem and return dual variables*

Description

Solves the linear assignment problem and returns dual potentials (u, v) in addition to the optimal matching. The dual variables provide an optimality certificate and enable sensitivity analysis.

Usage

```
assignment_duals(cost, maximize = FALSE)
```

Arguments

cost	Numeric matrix; rows = tasks, columns = agents. NA or Inf entries are treated as forbidden assignments.
maximize	Logical; if TRUE, maximizes the total cost instead of minimizing.

Details

The dual variables satisfy the complementary slackness conditions:

- For minimization: $u[i] + v[j] <= cost[i,j]$ for all (i,j)
- For any assigned pair (i,j): $u[i] + v[j] = cost[i,j]$

This implies that $sum(u) + sum(v) = total\_cost$ (strong duality).

Applications of dual variables:

- **Optimality verification**: Check that duals satisfy constraints
- **Sensitivity analysis**: Reduced cost $c[i,j] - u[i] - v[j]$ shows how much an edge cost must decrease before it enters the solution
- **Pricing in column generation**: Use duals to price new columns
- **Warm starting**: Reuse duals when costs change slightly

Value

A list with class "assignment_duals_result" containing:

- match - integer vector of column assignments (1-based)
- total_cost - optimal objective value
- u - numeric vector of row dual variables (length n)
- v - numeric vector of column dual variables (length m)
- status - character, e.g. "optimal"

See Also

assignment() for standard assignment without duals

Examples

```
cost <- matrix(c(4, 2, 5, 3, 3, 6, 7, 5, 4), nrow = 3, byrow = TRUE)
result <- assignment_duals(cost)

# Check optimality: u + v should equal cost for assigned pairs
for (i in 1:3) {
  j <- result$match[i]
  cat(sprintf("Row %d -> Col %d: u + v = %.2f, cost = %.2f\n",
              i, j, result$u[i] + result$v[j], cost[i, j]))
}

# Verify strong duality
cat("sum(u) + sum(v) =", sum(result$u) + sum(result$v), "\n")
cat("total_cost =", result$total_cost, "\n")

# Reduced costs (how much must cost decrease to enter solution)
reduced <- outer(result$u, result$v, "+")
reduced_cost <- cost - reduced
print(round(reduced_cost, 2))
```

---

as_assignment_matrix   *Convert assignment result to a binary matrix*

Description

Turns a tidy assignment result back into a 0/1 assignment matrix.

Usage

```
as_assignment_matrix(x, n_sources = NULL, n_targets = NULL)
```

Arguments

x	An assignment result object of class lap_solve_result
n_sources	Number of source nodes, optional
n_targets	Number of target nodes, optional

Value

Integer matrix with 0 and 1 entries

---

augment *Generic Augment Function*

Description

S3 generic for augmenting model results with original data.

Usage

```
augment(x, ...)
```

Arguments

x	An object to augment
...	Additional arguments passed to methods

Value

Augmented data (depends on method)

---

augment.matching_result

Augment Matching Results with Original Data (broom-style)

Description

S3 method for augmenting matching results following the broom package conventions. This is a thin wrapper around join_matched() with sensible defaults for quick exploration.

Usage

```
## S3 method for class 'matching_result'
augment(x, left, right, ...)
```

Arguments

x	A matching_result object
left	The original left dataset
right	The original right dataset
...	Additional arguments passed to join_matched()

Details

This method follows the `augment()` convention from the broom package, making it easy to integrate couplr into tidymodels workflows. It's equivalent to calling `join_matched()` with default parameters.

If the broom package is not loaded, you can use `couplr::augment()` to access this function.

Value

A tibble with matched pairs and original data (see `join_matched()`)

Examples

```
left <- data.frame(
  id = 1:5,
  treatment = 1,
  age = c(25, 30, 35, 40, 45)
)

right <- data.frame(
  id = 6:10,
  treatment = 0,
  age = c(24, 29, 36, 41, 44)
)

result <- match_couples(left, right, vars = "age")
couplr::augment(result, left, right)
```

---

autoplot.balance_diagnostics
ggplot2 autoplot for balance diagnostics

Description

Produces ggplot2-based balance assessment plots. Returns a ggplot object.

Usage

```
## S3 method for class 'balance_diagnostics'
autoplot(
  object,
  type = c("love", "histogram", "variance"),
  threshold = 0.1,
  ...
)
```

Arguments

`object`	A balance_diagnostics object
`type`	Type of plot: "love" (default), "histogram", or "variance"
`threshold`	Threshold for standardized differences (default: 0.1)
`...`	Additional arguments (ignored)

Value

A ggplot object

Examples

```
if (requireNamespace("ggplot2", quietly = TRUE)) {
  set.seed(42)
  left <- data.frame(id = 1:10, age = rnorm(10, 45, 10),
                     income = rnorm(10, 50000, 15000))
  right <- data.frame(id = 11:30, age = rnorm(20, 47, 10),
                      income = rnorm(20, 52000, 15000))
  result <- match_couples(left, right, vars = c("age", "income"))
  bal <- balance_diagnostics(result, left, right, vars = c("age", "income"))
  ggplot2::autoplot(bal)
}
```

---

autoplot.matching_result
ggplot2 autoplot for matching results

Description

Produces ggplot2-based visualizations of matching distance distributions. Returns a ggplot object that can be further customized.

Usage

```
## S3 method for class 'matching_result'
autoplot(object, type = c("histogram", "density", "ecdf"), ...)
```

Arguments

`object`	A matching_result object
`type`	Type of plot: "histogram" (default), "density", or "ecdf"
`...`	Additional arguments (ignored)

Details

Use `plot()` for base graphics or `autoplot()` for ggplot2 output. The ggplot2 package must be installed.

Value

A ggplot object

Examples

```
if (requireNamespace("ggplot2", quietly = TRUE)) {
  left <- data.frame(id = 1:5, x = c(1, 2, 3, 4, 5))
  right <- data.frame(id = 6:10, x = c(1.1, 2.2, 3.1, 4.2, 5.1))
  result <- match_couples(left, right, vars = "x")
  ggplot2::autoplot(result)
  ggplot2::autoplot(result, type = "density")
}
```

---

autoplot.sensitivity_analysis

ggplot2 autoplot for sensitivity analysis

Description

Plots p-value upper bounds against sensitivity parameter Gamma.

Usage

```
## S3 method for class 'sensitivity_analysis'
autoplot(object, alpha = 0.05, ...)
```

Arguments

object	A sensitivity_analysis object
alpha	Significance level (default: 0.05)
...	Additional arguments (ignored)

Value

A ggplot object

---

balance_diagnostics   *Balance Diagnostics for Matched Pairs*

Description

Computes comprehensive balance statistics comparing the distribution of matching variables between left and right units in the matched sample.

Usage

```
balance_diagnostics(
  result,
  left,
  right,
  vars = NULL,
  left_id = "id",
  right_id = "id"
)
```

Arguments

result	A matching result object from `match_couples()` or `greedy_couples()`
left	Data frame of left units
right	Data frame of right units
vars	Character vector of variable names to check balance for. Defaults to the variables used in matching (if available in result).
left_id	Character, name of ID column in left data (default: "id")
right_id	Character, name of ID column in right data (default: "id")

Details

This function computes several balance metrics:

Standardized Difference: The difference in means divided by the pooled standard deviation. Values less than 0.1 indicate excellent balance, 0.1-0.25 good balance.

Variance Ratio: The ratio of standard deviations (left/right). Values close to 1 are ideal.

KS Statistic: Kolmogorov-Smirnov test statistic comparing distributions. Lower values indicate more similar distributions.

Overall Metrics include mean absolute standardized difference across all variables, proportion of variables with large imbalance (|std diff| > 0.25), and maximum standardized difference.

Value

An S3 object of class `balance_diagnostics` containing:

var_stats Tibble with per-variable balance statistics

overall List with overall balance metrics

pairs Tibble of matched pairs with variables

n_matched Number of matched pairs

n_unmatched_left Number of unmatched left units

n_unmatched_right Number of unmatched right units

method Matching method used

has_blocks Whether blocking was used

block_stats Per-block statistics (if blocking used)

Examples

```
# Create sample data
set.seed(123)
left <- data.frame(
  id = 1:10,
  age = rnorm(10, 45, 10),
  income = rnorm(10, 50000, 15000)
)
right <- data.frame(
  id = 11:30,
  age = rnorm(20, 47, 10),
  income = rnorm(20, 52000, 15000)
)

# Match
result <- match_couples(left, right, vars = c("age", "income"))

# Get balance diagnostics
balance <- balance_diagnostics(result, left, right, vars = c("age", "income"))
print(balance)

# Get balance table
balance_table(balance)
```

---

balance_table   *Create Balance Table*

Description

Formats balance diagnostics into a clean table for display or export.

Usage

```
balance_table(balance, digits = 3)
```

Arguments

balance	A balance_diagnostics object from `balance_diagnostics()`
digits	Number of decimal places for rounding (default: 3)

Value

A tibble with formatted balance statistics

---

bottleneck_assignment   *Solve the Bottleneck Assignment Problem*

Description

Finds an assignment that minimizes (or maximizes) the maximum edge cost in a perfect matching. Unlike standard LAP which minimizes the sum of costs, BAP minimizes the maximum (bottleneck) cost.

Usage

```
bottleneck_assignment(cost, maximize = FALSE)
```

Arguments

cost	Numeric matrix; rows = tasks, columns = agents. `NA` or `Inf` entries are treated as forbidden assignments.
maximize	Logical; if `TRUE`, maximizes the minimum edge cost instead of minimizing the maximum (maximin objective). Default is `FALSE` (minimax).

Details

The Bottleneck Assignment Problem (BAP) is a variant of the Linear Assignment Problem where instead of minimizing the sum of assignment costs, we minimize the maximum cost among all assignments (minimax objective).

Algorithm: Uses binary search on the sorted unique costs combined with 'Hopcroft-Karp' bipartite matching to find the minimum threshold that allows a perfect matching.

Complexity: O(E * sqrt(V) * log(unique costs)) where E = edges, V = vertices.

Applications:

- Task scheduling with deadline constraints (minimize latest completion)
- Resource allocation (minimize maximum load/distance)
- Network routing (minimize maximum link utilization)
- Fair division problems (minimize maximum disparity)

Value

A list with class "bottleneck_result" containing:

- match - integer vector of length nrow(cost) giving the assigned column for each row (1-based indexing)

- bottleneck - numeric scalar, the bottleneck (max/min edge) value

- status - character scalar, e.g. "optimal"

See Also

assignment() for standard LAP (sum objective), lap_solve() for tidy LAP interface

Examples

```
# Simple example: minimize max cost
cost <- matrix(c(1, 5, 3,
                 2, 4, 6,
                 7, 1, 2), nrow = 3, byrow = TRUE)
result <- bottleneck_assignment(cost)
result$bottleneck  # Maximum edge cost in optimal assignment

# Maximize minimum (fair allocation)
profits <- matrix(c(10, 5, 8,
                     6, 12, 4,
                     3, 7, 11), nrow = 3, byrow = TRUE)
result <- bottleneck_assignment(profits, maximize = TRUE)
result$bottleneck  # Minimum profit among all assignments

# With forbidden assignments
cost <- matrix(c(1, NA, 3,
                 2, 4, Inf,
                 5, 1, 2), nrow = 3, byrow = TRUE)
result <- bottleneck_assignment(cost)
```

---

cardinality_match *Cardinality Matching*

Description

Maximizes the number of matched pairs subject to balance constraints. Uses iterative pruning: starts with a full match, then removes pairs that contribute most to imbalance until all variables satisfy the standardized difference threshold.

Usage

```
cardinality_match(
  left,
  right,
  vars,
  max_std_diff = 0.1,
  distance = "euclidean",
  weights = NULL,
  scale = FALSE,
  auto_scale = FALSE,
  method = "auto",
  max_iter = 100L,
  batch_fraction = 0.1
)
```

Arguments

left	Data frame of "left" units
right	Data frame of "right" units
vars	Character vector of matching variable names
max_std_diff	Maximum allowed absolute standardized difference (default: 0.1, corresponding to "excellent" balance)
distance	Distance metric (default: "euclidean")
weights	Optional named vector of variable weights
scale	Scaling method (default: FALSE)
auto_scale	If TRUE, automatically select scaling (default: FALSE)
method	LAP solver method (default: "auto")
max_iter	Maximum pruning iterations (default: 100)
batch_fraction	Fraction of worst pairs to remove per iteration (default: 0.1). Larger values speed up convergence but may over-prune.

Details

Cardinality matching (Zubizarreta 2014) finds the largest matched sample that satisfies pre-specified balance constraints. This implementation uses an iterative pruning heuristic:

1. Run full optimal matching via match_couples()
2. Compute balance diagnostics
3. While any |std_diff| exceeds max_std_diff:
   - Identify the variable with worst balance
   - Remove the batch of pairs contributing most to that imbalance
   - Recompute balance

Value

A matching_result object with additional cardinality info: `result$info$pruning_iterations`, `result$info$pairs_removed`, `result$info$final_balance`.

Examples

```
set.seed(42)
left <- data.frame(id = 1:20, x = rnorm(20, 0, 1), y = rnorm(20, 0, 1))
right <- data.frame(id = 21:50, x = rnorm(30, 0.5, 1), y = rnorm(30, 0.3, 1))
result <- cardinality_match(left, right, vars = c("x", "y"), max_std_diff = 0.2)
print(result)
```

---

compute_distances   *Compute and Cache Distance Matrix for Reuse*

Description

Precomputes a distance matrix between left and right datasets, allowing it to be reused across multiple matching operations with different constraints. This is particularly useful when exploring different matching parameters (max_distance, calipers, methods) without recomputing distances.

Usage

```
compute_distances(
  left,
  right,
  vars,
  distance = "euclidean",
  weights = NULL,
  scale = FALSE,
  auto_scale = FALSE,
  left_id = "id",
  right_id = "id",
  block_id = NULL
)
```

Arguments

left	Left dataset (data frame)
right	Right dataset (data frame)
vars	Character vector of variable names to use for distance computation
distance	Distance metric (default: "euclidean")
weights	Optional numeric vector of variable weights
scale	Scaling method: FALSE, "standardize", "range", or "robust"

auto_scale	Apply automatic preprocessing (default: FALSE)
left_id	Name of ID column in left (default: "id")
right_id	Name of ID column in right (default: "id")
block_id	Optional block ID column name for blocked matching

Details

This function computes distances once and stores them in a reusable object. The resulting distance_object can be passed to match_couples() or greedy_couples() instead of providing datasets and variables.

Benefits:

- **Performance**: Avoid recomputing distances when trying different constraints
- **Exploration**: Quickly test max_distance, calipers, or methods
- **Consistency**: Ensures same distances used across comparisons
- **Memory efficient**: Can use sparse matrices when many pairs are forbidden

The distance_object stores the original datasets, allowing downstream functions like join_matched() to work seamlessly.

Value

An S3 object of class "distance_object" containing:

- cost_matrix: Numeric matrix of distances
- left_ids: Character vector of left IDs
- right_ids: Character vector of right IDs
- block_id: Block ID column name (if specified)
- metadata: List with computation details (vars, distance, scale, etc.)
- original_left: Original left dataset (for later joining)
- original_right: Original right dataset (for later joining)

Examples

```
# Compute distances once
left <- data.frame(id = 1:5, age = c(25, 30, 35, 40, 45), income = c(45, 52, 48, 61, 55) * 1000)
right <- data.frame(id = 6:10, age = c(24, 29, 36, 41, 44), income = c(46, 51, 47, 60, 54) * 1000)

dist_obj <- compute_distances(
  left, right,
  vars = c("age", "income"),
  scale = "standardize"
)

# Reuse for different matching strategies
result1 <- match_couples(dist_obj, max_distance = 0.5)
result2 <- match_couples(dist_obj, max_distance = 1.0)
```

```
result3 <- greedy_couples(dist_obj, strategy = "sorted")

# All use the same precomputed distances
```

```
diagnose_distance_matrix
```
Diagnose distance matrix and suggest fixes

Description

Comprehensive diagnostics for a distance matrix with actionable suggestions.

Usage

```
diagnose_distance_matrix(
  cost_matrix,
  left = NULL,
  right = NULL,
  vars = NULL,
  warn = TRUE
)
```

Arguments

cost_matrix	Numeric matrix of distances
left	Left dataset (for variable checking)
right	Right dataset (for variable checking)
vars	Variables used for matching
warn	If TRUE, issue warnings

Value

List with diagnostic results and suggestions

---

example_costs   *Example cost matrices for assignment problems*

Description

Small example datasets for demonstrating couplr functionality across different assignment problem types: square, rectangular, sparse, and binary.

Usage

```
example_costs
```

Format

A list containing four example cost matrices:

simple_3x3 A 3x3 cost matrix with costs ranging from 2-7. Optimal assignment: row 1 -> col 2 (cost 2), row 2 -> col 1 (cost 3), row 3 -> col 3 (cost 4). Total optimal cost: 9.

rectangular_3x5 A 3x5 rectangular cost matrix demonstrating assignment when rows < columns. Each of 3 rows is assigned to one of 5 columns; 2 columns remain unassigned. Costs range 1-6.

sparse_with_na A 3x3 matrix with NA values indicating forbidden assignments. Use this to test algorithms' handling of constraints. Position (1,3), (2,2), and (3,1) are forbidden.

binary_costs A 3x3 matrix with binary (0/1) costs, suitable for testing the HK01 algorithm. Diagonal entries are 0 (preferred), off-diagonal entries are 1 (penalty).

Details

These matrices are designed to test different aspects of LAP solvers:

simple_3x3: Basic functionality test. Any correct solver should find total cost = 9.

rectangular_3x5: Tests handling of non-square problems. The optimal solution assigns all 3 rows with minimum total cost.

sparse_with_na: Tests constraint handling. Algorithms must avoid NA positions while finding an optimal assignment among valid entries.

binary_costs: Tests specialized binary cost algorithms. The optimal assignment uses all diagonal entries (total cost = 0).

See Also

lap_solve, example_df

Examples

```
# Simple 3x3 assignment
result <- lap_solve(example_costs$simple_3x3)
print(result)
# Optimal: sources 1,2,3 -> targets 2,1,3 with cost 9

# Rectangular problem (3 sources, 5 targets)
result <- lap_solve(example_costs$rectangular_3x5)
print(result)
# All 3 sources assigned; 2 targets unassigned

# Sparse problem with forbidden assignments
result <- lap_solve(example_costs$sparse_with_na)
print(result)
# Avoids NA positions

# Binary costs - test HK01 algorithm
result <- lap_solve(example_costs$binary_costs, method = "hk01")
print(result)
# Finds diagonal assignment (cost = 0)
```

---

example_df *Example assignment problem data frame*

Description

A tidy data frame representation of assignment problems, suitable for use with grouped workflows and batch solving. Contains two independent 3x3 assignment problems in long format.

Usage

```
example_df
```

Format

A tibble with 18 rows and 4 columns:

sim Simulation/problem identifier. Integer with values 1 or 2, distinguishing two independent assignment problems. Use with group_by(sim) for grouped solving.

source Source node index. Integer 1-3 representing the row (source) in each 3x3 cost matrix.

target Target node index. Integer 1-3 representing the column (target) in each 3x3 cost matrix.

cost Cost of assigning source to target. Numeric values ranging from 1-7. Each source-target pair has exactly one cost entry.

Details

This dataset demonstrates couplr's data frame interface for LAP solving. The long format (one row per source-target pair) is converted internally to a cost matrix for solving.

Simulation 1: Costs from example_costs$simple_3x3

- Optimal assignment: (1->2, 2->1, 3->3)
- Total cost: 9

Simulation 2: Different cost structure

- Optimal assignment: (1->1, 2->3, 3->3) or equivalent
- Total cost: 4

See Also

lap_solve, lap_solve_batch, example_costs

Examples

```
library(dplyr)

# Solve both problems with grouped workflow
example_df |>
  group_by(sim) |>
  lap_solve(source, target, cost)

# Batch solving for efficiency
example_df |>
  group_by(sim) |>
  lap_solve_batch(source, target, cost)

# Inspect the data structure
example_df |>
  group_by(sim) |>
  summarise(
    n_pairs = n(),
    min_cost = min(cost),
    max_cost = max(cost)
  )
```

| get_method_used | *Extract method used from assignment result* |

Description

Extract method used from assignment result

Usage

```
get_method_used(x)
```

Arguments

x	An assignment result object

Value

Character string indicating method used

---

get_total_cost *Extract total cost from assignment result*

Description

Extract total cost from assignment result

Usage

```
get_total_cost(x)
```

Arguments

x	An assignment result object

Value

Numeric total cost

---

greedy_couples *Fast approximate matching using greedy algorithm*

Description

Performs fast one-to-one matching using greedy strategies. Does not guarantee optimal total distance but is much faster than [match_couples()](#) for large datasets. Supports blocking, distance constraints, and various distance metrics.

Usage

```
greedy_couples(
  left,
  right = NULL,
  vars = NULL,
  distance = "euclidean",
  weights = NULL,
  scale = FALSE,
  auto_scale = FALSE,
  max_distance = Inf,
  calipers = NULL,
  block_id = NULL,
  ignore_blocks = FALSE,
  require_full_matching = FALSE,
  strategy = c("row_best", "sorted", "pq"),
  return_unmatched = TRUE,
  return_diagnostics = FALSE,
  parallel = FALSE,
  replace = FALSE,
  ratio = 1L,
  check_costs = TRUE
)
```

Arguments

`left`	Data frame of "left" units (e.g., treated, cases)
`right`	Data frame of "right" units (e.g., control, controls)
`vars`	Variable names to use for distance computation
`distance`	Distance metric: "euclidean", "manhattan", "mahalanobis", or a custom function
`weights`	Optional named vector of variable weights
`scale`	Scaling method: FALSE (none), "standardize", "range", or "robust"
`auto_scale`	If TRUE, automatically check variable health and select scaling method (default: FALSE)
`max_distance`	Maximum allowed distance (pairs exceeding this are forbidden)
`calipers`	Named list of per-variable maximum absolute differences
`block_id`	Column name containing block IDs (for stratified matching)
`ignore_blocks`	If TRUE, ignore block_id even if present
`require_full_matching`	
	If TRUE, error if any units remain unmatched
`strategy`	Greedy strategy:

- "row_best": For each row, find best available column (default)
- "sorted": Sort all pairs by distance, greedily assign
- "pq": Use priority queue (good for very large problems)

```
return_unmatched
                Include unmatched units in output
return_diagnostics
                Include detailed diagnostics in output
```

parallel	Enable parallel processing for blocked matching. Requires 'future' and 'future.apply' packages. Can be:

- FALSE: Sequential processing (default)
- TRUE: Auto-configure parallel backend
- Character: Specify future plan (e.g., "multisession", "multicore")

replace	If TRUE, allow matching with replacement (same right unit can be matched to multiple left units). Default: FALSE.
ratio	Integer, number of right units to match per left unit. Default: 1 (one-to-one matching). For k:1 matching, set ratio = k.
check_costs	If TRUE, check distance distribution for potential problems and provide helpful warnings before matching (default: TRUE)

Details

Greedy strategies do not guarantee optimal total distance but are much faster:

- "row_best": O(n*m) time, simple and often produces good results
- "sorted": O(n$m$log(n*m)) time, better quality but slower
- "pq": O(n$m$log(n*m)) time, memory-efficient for large problems

Use greedy_couples when:

- Dataset is very large (> 10,000 x 10,000)
- Approximate solution is acceptable
- Speed is more important than optimality

Value

A list with class "matching_result" (same structure as match_couples)

Examples

```
# Basic greedy matching
left <- data.frame(id = 1:100, x = rnorm(100))
right <- data.frame(id = 101:200, x = rnorm(100))
result <- greedy_couples(left, right, vars = "x")

# Compare to optimal
result_opt <- match_couples(left, right, vars = "x")
result_greedy <- greedy_couples(left, right, vars = "x")
result_greedy$info$total_distance / result_opt$info$total_distance  # Quality ratio
```

| hospital_staff | *Hospital staff scheduling example dataset* |

Description

A comprehensive example dataset for demonstrating couplr functionality across vignettes. Contains hospital staff scheduling data with nurses, shifts, costs, and preference scores suitable for assignment problems, as well as nurse characteristics for matching workflows.

Usage

```
hospital_staff
```

Format

A list containing eight related datasets:

basic_costs A 10x10 numeric cost matrix for assigning 10 nurses to 10 shifts. Values range from approximately 1-15, where lower values indicate better fit (less overtime, matches skills, respects preferences). Use with lap_solve() for basic assignment.

preferences A 10x10 numeric preference matrix on a 0-10 scale, where higher values indicate stronger nurse preference for a shift. Use with lap_solve(..., maximize = TRUE) to optimize preferences rather than minimize costs.

schedule_df A tibble with 100 rows (10 nurses x 10 shifts) in long format for data frame workflows:

nurse_id Integer 1-10. Unique identifier for each nurse.

shift_id Integer 1-10. Unique identifier for each shift.

cost Numeric. Assignment cost (same values as basic_costs).

preference Numeric 0-10. Nurse preference score.

skill_match Integer 0/1. Binary indicator: 1 if nurse skills match shift requirements, 0 otherwise.

nurses A tibble with 10 rows describing nurse characteristics:

nurse_id Integer 1-10. Links to schedule_df and basic_costs rows.

experience_years Numeric 1-20. Years of nursing experience.

department Character. Primary department: "ICU", "ER", "General", or "Pediatrics".

shift_preference Character. Preferred shift type: "day", "evening", or "night".

certification_level Integer 1-3. Certification level where 3 is highest (e.g., 1=RN, 2=BSN, 3=MSN).

shifts A tibble with 10 rows describing shift requirements:

shift_id Integer 1-10. Links to schedule_df and basic_costs cols.

department Character. Department needing coverage.

shift_type Character. Shift type: "day", "evening", or "night".

min_experience Numeric. Minimum years of experience required.

> **min_certification** Integer 1-3. Minimum certification level.

weekly_df A tibble for batch solving with 500 rows (5 days x 10 nurses x 10 shifts):

> **day** Character. Day of week: "Mon", "Tue", "Wed", "Thu", "Fri".
>
> **nurse_id** Integer 1-10. Nurse identifier.
>
> **shift_id** Integer 1-10. Shift identifier.
>
> **cost** Numeric. Daily assignment cost (varies by day).
>
> **preference** Numeric 0-10. Daily preference score.
>
> Use with `group_by(day)` for solving each day's schedule.

nurses_extended A tibble with 200 nurses for matching examples, representing a treatment group (e.g., full-time nurses):

> **nurse_id** Integer 1-200. Unique identifier.
>
> **age** Numeric 22-65. Nurse age in years.
>
> **experience_years** Numeric 0-40. Years of nursing experience.
>
> **hourly_rate** Numeric 25-75. Hourly wage in dollars.
>
> **department** Character. Primary department assignment.
>
> **certification_level** Integer 1-3. Certification level.
>
> **is_fulltime** Logical. TRUE for full-time status.

controls_extended A tibble with 300 potential control nurses (e.g., part-time or registry nurses) for matching. Same structure as nurses_extended. Designed to have systematic differences from nurses_extended (older, less experience on average) to demonstrate matching's ability to create comparable groups.

Details

This dataset is used throughout the couplr documentation to provide a consistent, realistic example that evolves in complexity. It supports three use cases: (1) basic LAP solving with cost matrices, (2) batch solving across multiple days, and (3) matching workflows comparing nurse groups.

The dataset is designed to demonstrate progressively complex scenarios:

Basic LAP (`vignette("getting-started")`):

- `basic_costs`: Simple 10x10 assignment
- `preferences`: Maximization problem
- `schedule_df`: Data frame input, grouped workflows
- `weekly_df`: Batch solving across days

Algorithm comparison (`vignette("algorithms")`):

- Use `basic_costs` to compare algorithm behavior
- Modify with NA values for sparse scenarios

Matching workflows (`vignette("matching-workflows")`):

- `nurses_extended`: Treatment group (full-time nurses)
- `controls_extended`: Control pool (part-time/registry nurses)
- Match on age, experience, department for causal analysis

See Also

lap_solve for basic assignment solving, lap_solve_batch for batch solving, match_couples for matching workflows, vignette("getting-started") for introductory tutorial

Examples

```
# Basic assignment: assign nurses to shifts minimizing cost
lap_solve(hospital_staff$basic_costs)

# Maximize preferences instead
lap_solve(hospital_staff$preferences, maximize = TRUE)

# Data frame workflow
library(dplyr)
hospital_staff$schedule_df |>
  lap_solve(nurse_id, shift_id, cost)

# Batch solve weekly schedule
hospital_staff$weekly_df |>
  group_by(day) |>
  lap_solve(nurse_id, shift_id, cost)

# Matching workflow: match full-time to part-time nurses
match_couples(
  left = hospital_staff$nurses_extended,
  right = hospital_staff$controls_extended,
  vars = c("age", "experience_years", "certification_level"),
  auto_scale = TRUE
)
```

---

is_distance_object   *Check if Object is a Distance Object*

Description

Check if Object is a Distance Object

Usage

```
is_distance_object(x)
```

Arguments

x   Object to check

Value

Logical: TRUE if x is a distance_object

Examples

```
left <- data.frame(id = 1:3, x = c(1, 2, 3))
right <- data.frame(id = 4:6, x = c(1.1, 2.1, 3.1))
dist_obj <- compute_distances(left, right, vars = "x")
is_distance_object(dist_obj)  # TRUE
is_distance_object(list())    # FALSE
```

---

is_lap_solve_batch_result

Check if object is a batch assignment result

Description

Check if object is a batch assignment result

Usage

```
is_lap_solve_batch_result(x)
```

Arguments

x   Object to test

Value

Logical indicating if x is a batch assignment result

---

is_lap_solve_kbest_result

Check if object is a k-best assignment result

Description

Check if object is a k-best assignment result

Usage

```
is_lap_solve_kbest_result(x)
```

Arguments

x   Object to test

Value

Logical indicating if x is a k-best assignment result

---

is_lap_solve_result *Check if object is an assignment result*

Description

Check if object is an assignment result

Usage

```
is_lap_solve_result(x)
```

Arguments

x   Object to test

Value

Logical indicating if x is an assignment result

---

join_matched   *Join Matched Pairs with Original Data*

Description

Creates an analysis-ready dataset by joining matched pairs with variables from the original left and right datasets. This eliminates the need for manual joins and provides a convenient format for downstream analysis.

Usage

```
join_matched(
  result,
  left,
  right,
  left_vars = NULL,
  right_vars = NULL,
  left_id = "id",
  right_id = "id",
  suffix = c("_left", "_right"),
  include_distance = TRUE,
  include_pair_id = TRUE,
  include_block_id = TRUE
)
```

Arguments

`result`	A matching_result object from `match_couples()` or `greedy_couples()`
`left`	The original left dataset
`right`	The original right dataset
`left_vars`	Character vector of variable names to include from left. If NULL (default), includes all variables except the ID column.
`right_vars`	Character vector of variable names to include from right. If NULL (default), includes all variables except the ID column.
`left_id`	Name of the ID column in left dataset (default: "id")
`right_id`	Name of the ID column in right dataset (default: "id")
`suffix`	Character vector of length 2 specifying suffixes for left and right variables (default: c("_left", "_right"))
`include_distance`	
	Include the matching distance in output (default: TRUE)
`include_pair_id`	
	Include pair_id column (default: TRUE)
`include_block_id`	
	Include block_id if blocking was used (default: TRUE)

Details

This function simplifies the common workflow of joining matched pairs with original data. Instead of manually merging result$pairs with left and right datasets, `join_matched()` handles the joins automatically and applies consistent naming conventions.

When variables appear in both left and right datasets, suffixes are appended to distinguish them (e.g., "age_left" and "age_right"). This makes it easy to compute differences or use both values in models.

Value

A tibble with one row per matched pair, containing:

- `pair_id`: Sequential pair identifier (if include_pair_id = TRUE)
- `left_id`: ID from left dataset
- `right_id`: ID from right dataset
- `distance`: Matching distance (if include_distance = TRUE)
- `block_id`: Block identifier (if blocking used and include_block_id = TRUE)
- Variables from left dataset (with left suffix)
- Variables from right dataset (with right suffix)

Examples

```
# Basic usage
left <- data.frame(
  id = 1:5,
  treatment = 1,
  age = c(25, 30, 35, 40, 45),
  income = c(45000, 52000, 48000, 61000, 55000)
)

right <- data.frame(
  id = 6:10,
  treatment = 0,
  age = c(24, 29, 36, 41, 44),
  income = c(46000, 51500, 47500, 60000, 54000)
)

result <- match_couples(left, right, vars = c("age", "income"))
matched_data <- join_matched(result, left, right)
head(matched_data)

# Specify which variables to include
matched_data <- join_matched(
  result, left, right,
  left_vars = c("treatment", "age", "income"),
  right_vars = c("age", "income"),
  suffix = c("_treated", "_control")
)

# Without distance or pair_id
matched_data <- join_matched(
  result, left, right,
  include_distance = FALSE,
  include_pair_id = FALSE
)
```

---

lap_solve   *Solve linear assignment problems*

Description

Provides a tidy interface for solving the linear assignment problem using 'Hungarian' or 'Jonker-Volgenant' algorithms. Supports rectangular matrices, NA/Inf masking, and data frame inputs.

Usage

```
lap_solve(
  x,
  source = NULL,
```

```
    target = NULL,
    cost = NULL,
    maximize = FALSE,
    method = "auto",
    forbidden = NA
)
```

Arguments

x	Cost matrix, data frame, or tibble. If a data frame/tibble, must include columns specified by source, target, and cost.
source	Column name for source/row indices (if x is a data frame)
target	Column name for target/column indices (if x is a data frame)
cost	Column name for costs (if x is a data frame)
maximize	Logical; if TRUE, maximizes total cost instead of minimizing (default: FALSE)
method	Algorithm to use. One of:

- "auto" (default): automatically selects best algorithm
- "jv": 'Jonker-Volgenant' algorithm (general purpose, fast)
- "hungarian": Classic 'Hungarian' algorithm
- "auction": 'Bertsekas' auction algorithm (good for large dense problems)
- "sap": Sparse assignment (good for sparse/rectangular problems)
- "hk01": 'Hopcroft-Karp' for binary/uniform costs

forbidden	Value to mark forbidden assignments (default: NA). Can also use Inf.

Value

A tibble with columns:

- source: row/source indices
- target: column/target indices
- cost: cost of each assignment
- total_cost: total cost (attribute)

Examples

```
# Matrix input
cost <- matrix(c(4, 2, 5, 3, 3, 6, 7, 5, 4), nrow = 3)
lap_solve(cost)

# Data frame input
library(dplyr)
df <- tibble(
  source = rep(1:3, each = 3),
  target = rep(1:3, times = 3),
  cost = c(4, 2, 5, 3, 3, 6, 7, 5, 4)
)
lap_solve(df, source, target, cost)
```

```
# With NA masking (forbidden assignments)
cost[1, 3] <- NA
lap_solve(cost)

# Grouped data frames
df <- tibble(
  sim = rep(1:2, each = 9),
  source = rep(1:3, times = 6),
  target = rep(1:3, each = 3, times = 2),
  cost = runif(18, 1, 10)
)
df |> group_by(sim) |> lap_solve(source, target, cost)
```

---

lap_solve_batch   *Solve multiple assignment problems efficiently*

Description

Solve many independent assignment problems at once. Supports lists of matrices, 3D arrays, or grouped data frames. Optional parallel execution via n_threads.

Usage

```
lap_solve_batch(
  x,
  source = NULL,
  target = NULL,
  cost = NULL,
  maximize = FALSE,
  method = "auto",
  n_threads = 1,
  forbidden = NA
)
```

Arguments

x	One of: List of cost matrices, 3D array, or grouped data frame
source	Column name for source indices (if x is a grouped data frame)
target	Column name for target indices (if x is a grouped data frame)
cost	Column name for costs (if x is a grouped data frame)
maximize	Logical; if TRUE, maximizes total cost (default: FALSE)
method	Algorithm to use (default: "auto"). See lap_solve for options.
n_threads	Number of threads for parallel execution (default: 1). Set to NULL to use all available cores.
forbidden	Value to mark forbidden assignments (default: NA)

Value

A tibble with columns:

- `problem_id`: identifier for each problem
- `source`: source indices for assignments
- `target`: target indices for assignments
- `cost`: cost of each assignment
- `total_cost`: total cost for each problem
- `method_used`: algorithm used for each problem

Examples

```
# List of matrices
costs <- list(
  matrix(c(1, 2, 3, 4), 2, 2),
  matrix(c(5, 6, 7, 8), 2, 2)
)
lap_solve_batch(costs)

# 3D array
arr <- array(runif(2 * 2 * 10), dim = c(2, 2, 10))
lap_solve_batch(arr)

# Grouped data frame
library(dplyr)
df <- tibble(
  sim = rep(1:5, each = 9),
  source = rep(1:3, times = 15),
  target = rep(1:3, each = 3, times = 5),
  cost = runif(45, 1, 10)
)
df |> group_by(sim) |> lap_solve_batch(source, target, cost)

# Parallel execution (requires n_threads > 1)
lap_solve_batch(costs, n_threads = 2)
```

---

lap_solve_kbest   *Find k-best optimal assignments*

Description

Returns the top k optimal (or near-optimal) assignments using 'Murty' algorithm. Useful for exploring alternative optimal solutions or finding robust assignments.

Usage

```
lap_solve_kbest(
  x,
  k = 3,
  source = NULL,
  target = NULL,
  cost = NULL,
  maximize = FALSE,
  method = "murty",
  single_method = "jv",
  forbidden = NA
)
```

Arguments

x	Cost matrix, data frame, or tibble. If a data frame/tibble, must include columns specified by source, target, and cost.
k	Number of best solutions to return (default: 3)
source	Column name for source/row indices (if x is a data frame)
target	Column name for target/column indices (if x is a data frame)
cost	Column name for costs (if x is a data frame)
maximize	Logical; if TRUE, finds k-best maximizing assignments (default: FALSE)
method	Algorithm for each sub-problem (default: "murty"). Future versions may support additional methods.
single_method	Algorithm used for solving each node in the search tree (default: "jv")
forbidden	Value to mark forbidden assignments (default: NA)

Value

A tibble with columns:

- rank: ranking of solutions (1 = best, 2 = second best, etc.)
- solution_id: unique identifier for each solution
- source: source indices
- target: target indices
- cost: cost of each edge in the assignment
- total_cost: total cost of the complete solution

Examples

```
# Matrix input - find 5 best solutions
cost <- matrix(c(4, 2, 5, 3, 3, 6, 7, 5, 4), nrow = 3)
lap_solve_kbest(cost, k = 5)

# Data frame input
```

```
library(dplyr)
df <- tibble(
  source = rep(1:3, each = 3),
  target = rep(1:3, times = 3),
  cost = c(4, 2, 5, 3, 3, 6, 7, 5, 4)
)
lap_solve_kbest(df, k = 3, source, target, cost)

# With maximization
lap_solve_kbest(cost, k = 3, maximize = TRUE)
```

---

lap_solve_line_metric   *Solve 1-D Line Assignment Problem*

Description

Solves the linear assignment problem when both sources and targets are ordered points on a line. Uses efficient O(n*m) dynamic programming for rectangular problems and O(n) sorting for square problems.

Usage

```
lap_solve_line_metric(x, y, cost = "L1", maximize = FALSE)
```

Arguments

x	Numeric vector of source positions (will be sorted internally)
y	Numeric vector of target positions (will be sorted internally)
cost	Cost function for distance. Either:

- "L1" (default): absolute distance ('Manhattan' distance)
- "L2": squared distance (squared 'Euclidean' distance) Can also use aliases: "abs", "manhattan" for L1; "sq", "squared", "quadratic" for L2

maximize	Logical; if TRUE, maximizes total cost instead of minimizing (default: FALSE)

Details

This is a specialized solver that exploits the structure of 1-dimensional assignment problems where costs depend only on the distance between points on a line. It is much faster than general LAP solvers for this special case.

The algorithm works as follows:

Square case (n == m): Both vectors are sorted and matched in order: $x[1] \rightarrow y[1]$, $x[2] \rightarrow y[2]$, etc. This is optimal for any metric cost function on a line.

Rectangular case (n < m): Uses dynamic programming to find the optimal assignment that matches all n sources to a subset of the m targets, minimizing total distance. The DP recurrence is:

```
dp[i][j] = min(dp[i][j-1], dp[i-1][j-1] + cost(x[i], y[j]))
```

This finds the minimum cost to match the first i sources to the first j targets.

Complexity:

- Time: O(n*m) for rectangular, O(n log n) for square
- Space: O(n*m) for DP table

Value

A list with components:

- `match`: Integer vector of length n with 1-based column indices
- `total_cost`: Total cost of the assignment

Examples

```
# Square case: equal number of sources and targets
x <- c(1.5, 3.2, 5.1)
y <- c(2.0, 3.0, 5.5)
result <- lap_solve_line_metric(x, y, cost = "L1")
print(result)

# Rectangular case: more targets than sources
x <- c(1.0, 3.0, 5.0)
y <- c(0.5, 2.0, 3.5, 4.5, 6.0)
result <- lap_solve_line_metric(x, y, cost = "L2")
print(result)

# With unsorted inputs (will be sorted internally)
x <- c(5.0, 1.0, 3.0)
y <- c(4.5, 0.5, 6.0, 2.0, 3.5)
result <- lap_solve_line_metric(x, y, cost = "L1")
print(result)
```

---

matchmaker   *Create blocks for stratified matching*

Description

Constructs blocks (strata) for matching, using either grouping variables or clustering algorithms. Returns the input data frames with block IDs assigned, along with block summary statistics.

Usage

```
matchmaker(
  left,
  right,
  block_type = c("none", "group", "cluster"),
  block_by = NULL,
  block_vars = NULL,
  block_method = "kmeans",
  n_blocks = NULL,
  min_left = 1,
  min_right = 1,
  drop_imbalanced = FALSE,
  imbalance_threshold = Inf,
  return_dropped = TRUE,
  ...
)
```

Arguments

`left`	Data frame of "left" units (e.g., treated, cases)
`right`	Data frame of "right" units (e.g., control, controls)
`block_type`	Type of blocking to use:

- "none": No blocking (default)
- "group": Block by existing categorical variable(s)
- "cluster": Block using clustering algorithm

`block_by`	Variable name(s) for grouping (if block_type = "group")
`block_vars`	Variable names for clustering (if block_type = "cluster")
`block_method`	Clustering method (if block_type = "cluster"):

- "kmeans": K-means clustering
- "hclust": Hierarchical clustering

`n_blocks`	Target number of blocks (for clustering)
`min_left`	Minimum number of left units per block
`min_right`	Minimum number of right units per block
`drop_imbalanced`	
	Drop blocks with extreme imbalance
`imbalance_threshold`	
	Maximum allowed |n_left - n_right| / max(n_left, n_right)
`return_dropped`	Include dropped blocks in output
`...`	Additional arguments passed to clustering function

Details

This function does NOT perform matching - it only creates the block structure. Use `match_couples()` or `greedy_couples()` to perform matching within blocks.

Value

A list with class "matchmaker_result" containing:

- `left`: Left data frame with block_id column added
- `right`: Right data frame with block_id column added
- `block_summary`: Summary statistics for each block
- `dropped`: Information about dropped blocks (if any)
- `info`: Metadata about blocking process

Examples

```
# Group blocking
left <- data.frame(id = 1:10, region = rep(c("A", "B"), each = 5), x = rnorm(10))
right <- data.frame(id = 11:20, region = rep(c("A", "B"), each = 5), x = rnorm(10))
blocks <- matchmaker(left, right, block_type = "group", block_by = "region")
print(blocks$block_summary)

# Clustering
blocks <- matchmaker(left, right, block_type = "cluster",
                     block_vars = "x", n_blocks = 3)
```

---

match_couples   *Optimal matching using linear assignment*

Description

Performs optimal one-to-one matching between two datasets using linear assignment problem (LAP) solvers. Supports blocking, distance constraints, and various distance metrics.

Usage

```
match_couples(
  left,
  right = NULL,
  vars = NULL,
  distance = "euclidean",
  weights = NULL,
  scale = FALSE,
  auto_scale = FALSE,
  max_distance = Inf,
  calipers = NULL,
  block_id = NULL,
  ignore_blocks = FALSE,
  require_full_matching = FALSE,
  method = "auto",
```

```
    return_unmatched = TRUE,
    return_diagnostics = FALSE,
    parallel = FALSE,
    replace = FALSE,
    ratio = 1L,
    check_costs = TRUE
)
```

Arguments

`left`	Data frame of "left" units (e.g., treated, cases)
`right`	Data frame of "right" units (e.g., control, controls)
`vars`	Variable names to use for distance computation
`distance`	Distance metric: "euclidean", "manhattan", "mahalanobis", or a custom function
`weights`	Optional named vector of variable weights
`scale`	Scaling method: FALSE (none), "standardize", "range", or "robust"
`auto_scale`	If TRUE, automatically check variable health and select scaling method (default: FALSE)
`max_distance`	Maximum allowed distance (pairs exceeding this are forbidden)
`calipers`	Named list of per-variable maximum absolute differences
`block_id`	Column name containing block IDs (for stratified matching)
`ignore_blocks`	If TRUE, ignore block_id even if present
`require_full_matching`	
	If TRUE, error if any units remain unmatched
`method`	LAP solver: "auto", "hungarian", "jv", "gabow_tarjan", etc.
`return_unmatched`	
	Include unmatched units in output
`return_diagnostics`	
	Include detailed diagnostics in output
`parallel`	Enable parallel processing for blocked matching. Requires 'future' and 'future.apply' packages. Can be: • FALSE: Sequential processing (default) • TRUE: Auto-configure parallel backend • Character: Specify future plan (e.g., "multisession", "multicore")
`replace`	If TRUE, allow matching with replacement (same right unit can be matched to multiple left units). Default: FALSE.
`ratio`	Integer, number of right units to match per left unit. Default: 1 (one-to-one matching). For k:1 matching, set ratio = k.
`check_costs`	If TRUE, check distance distribution for potential problems and provide helpful warnings before matching (default: TRUE)

Details

This function finds the matching that minimizes total distance among all feasible matchings, subject to constraints. Use [`greedy_couples()`](#) for faster approximate matching on large datasets.

Value

A list with class "matching_result" containing:

- `pairs`: Tibble of matched pairs with distances
- `unmatched`: List of unmatched left and right IDs
- `info`: Matching diagnostics and metadata

Examples

```
# Basic matching
left <- data.frame(id = 1:5, x = c(1, 2, 3, 4, 5), y = c(2, 4, 6, 8, 10))
right <- data.frame(id = 6:10, x = c(1.1, 2.2, 3.1, 4.2, 5.1), y = c(2.1, 4.1, 6.2, 8.1, 10.1))
result <- match_couples(left, right, vars = c("x", "y"))
print(result$pairs)

# With constraints
result <- match_couples(left, right, vars = c("x", "y"),
                        max_distance = 1,
                        calipers = list(x = 0.5))

# With blocking
left$region <- c("A", "A", "B", "B", "B")
right$region <- c("A", "A", "B", "B", "B")
blocks <- matchmaker(left, right, block_type = "group", block_by = "region")
result <- match_couples(blocks$left, blocks$right, vars = c("x", "y"))
```

---

pixel_morph   *Pixel-level image morphing (final frame only)*

Description

Computes optimal pixel assignment from A to B and returns the final transported frame (without intermediate animation frames).

Usage

```
pixel_morph(
  imgA,
  imgB,
  n_frames = 16L,
  mode = c("color_walk", "exact", "recursive"),
  lap_method = "jv",
  maximize = FALSE,
  quantize_bits = 5L,
  downscale_steps = 0L,
  alpha = 1,
  beta = 0,
```

```
    patch_size = 1L,
    upscale = 1,
    show = interactive()
)
```

Arguments

imgA	Source image (file path or magick image object)
imgB	Target image (file path or magick image object)
n_frames	Internal parameter for rendering (default: 16)
mode	Assignment algorithm: "color_walk" (default), "exact", or "recursive"
lap_method	LAP solver method (default: "jv")
maximize	Logical, maximize instead of minimize cost (default: FALSE)
quantize_bits	Color quantization for "color_walk" mode (default: 5)
downscale_steps	
	Number of 2x reductions before computing assignment (default: 0)
alpha	Weight for color distance in cost function (default: 1)
beta	Weight for spatial distance in cost function (default: 0)
patch_size	Tile size for tiled modes (default: 1)
upscale	Post-rendering upscaling factor (default: 1)
show	Logical, display result in viewer (default: interactive())

Details

Transport-Only Semantics:

This function returns a SHARP, pixel-perfect transport of A's pixels to positions determined by the assignment to B.

Key Points:

- Assignment computed using: `cost = alpha * color_dist + beta * spatial_dist`
- B's COLORS influence assignment but DO NOT appear in output
- Result has A's colors arranged to match B's layout
- No motion blur (unlike intermediate frames in animation)

See `pixel_morph_animate` for detailed explanation of assignment vs rendering semantics.

Permutation Warnings:

Assignment is guaranteed to be a bijection (permutation) ONLY when:

- `downscale_steps = 0` (no resolution changes)
- `mode = "exact"` with `patch_size = 1`

With downscaling or tiled modes, assignment may have:

- **Overlaps:** Multiple source pixels map to same destination (last write wins)
- **Holes:** Some destinations never filled (remain transparent)

If assignment is not a bijection (due to downscaling or tiling), a warning will be issued. The result may contain:

- Overlapped pixels (multiple sources -> one destination)
- Transparent holes (some destinations unfilled)

For guaranteed pixel-perfect results, use:

```
pixel_morph(A, B, mode = "exact", downscale_steps = 0)
```

Value

magick image object of the final transported frame

See Also

pixel_morph_animate for animated version

Examples

```
if (requireNamespace("magick", quietly = TRUE)) {
  imgA <- system.file("extdata/icons/circleA_40.png", package = "couplr")
  imgB <- system.file("extdata/icons/circleB_40.png", package = "couplr")
  if (nzchar(imgA) && nzchar(imgB)) {
    result <- pixel_morph(imgA, imgB, n_frames = 4, show = FALSE)
  }
}
```

---

pixel_morph_animate   *Pixel-level image morphing (animation)*

Description

Creates an animated morph by computing optimal pixel assignment from image A to image B, then rendering intermediate frames showing the transport.

Usage

```
pixel_morph_animate(
  imgA,
  imgB,
  n_frames = 16L,
  fps = 10L,
  format = c("gif", "webp", "mp4"),
  outfile = NULL,
  show = interactive(),
  mode = c("color_walk", "exact", "recursive"),
  lap_method = "jv",
  maximize = FALSE,
  quantize_bits = 5L,
  downscale_steps = 0L,
```

```
    alpha = 1,
    beta = 0,
    patch_size = 1L,
    upscale = 1
)
```

Arguments

imgA	Source image (file path or magick image object)
imgB	Target image (file path or magick image object)
n_frames	Integer number of animation frames (default: 16)
fps	Frames per second for playback (default: 10)
format	Output format: "gif", "webp", or "mp4"
outfile	Optional output file path
show	Logical, display animation in viewer (default: interactive())
mode	Assignment algorithm: "color_walk" (default), "exact", or "recursive"
lap_method	LAP solver method (default: "jv")
maximize	Logical, maximize instead of minimize cost (default: FALSE)
quantize_bits	Color quantization for "color_walk" mode (default: 5)
downscale_steps	
	Number of 2x reductions before computing assignment (default: 0)
alpha	Weight for color distance in cost function (default: 1)
beta	Weight for spatial distance in cost function (default: 0)
patch_size	Tile size for tiled modes (default: 1)
upscale	Post-rendering upscaling factor (default: 1)

Details

Assignment vs Rendering Semantics:

CRITICAL: This function has two separate phases with different semantics:

Phase 1 - Assignment Computation:

The assignment is computed by minimizing:

```
    cost(i,j) = alpha * color_distance(A[i], B[j]) +
                beta * spatial_distance(pos_i, pos_j)
```

This means B's COLORS influence which pixels from A map to which positions.

Phase 2 - Rendering (Transport-Only):

The renderer uses ONLY A's colors:

- Intermediate frames: A's pixels move along paths with motion blur
- Final frame: A's pixels at their assigned positions (sharp, no blur)
- B's colors NEVER appear in the output

Result: You get A's colors rearranged to match B's geometry/layout.

What This Means:

- B influences WHERE pixels go (via similarity in cost function)
- B does NOT determine WHAT COLORS appear in output
- Final image has A's palette arranged to mimic B's structure

Parameter Guidance:

For pure spatial rearrangement (ignore B's colors in assignment):

```
pixel_morph_animate(A, B, alpha = 0, beta = 1)
```

For color-similarity matching (default):

```
pixel_morph_animate(A, B, alpha = 1, beta = 0)
```

For hybrid (color + spatial):

```
pixel_morph_animate(A, B, alpha = 1, beta = 0.2)
```

Permutation Guarantees:

Assignment is guaranteed to be a bijection (permutation) ONLY when:

- downscale_steps = 0 (no resolution changes)
- mode = "exact" with patch_size = 1

With downscaling or tiled modes, assignment may have:

- **Overlaps:** Multiple source pixels map to same destination (last write wins)
- **Holes:** Some destinations never filled (remain transparent)

A warning is issued if overlaps/holes are detected in the final frame.

Value

Invisibly returns a list with animation object and metadata:

animation	magick animation object
width	Image width in pixels
height	Image height in pixels
assignment	Integer vector of 1-based assignment indices (R convention)
n_pixels	Total number of pixels
mode	Mode used for matching
upscale	Upscaling factor applied

Examples

```
if (requireNamespace("magick", quietly = TRUE)) {
  imgA <- system.file("extdata/icons/circleA_40.png", package = "couplr")
  imgB <- system.file("extdata/icons/circleB_40.png", package = "couplr")
  if (nzchar(imgA) && nzchar(imgB)) {
    outfile <- tempfile(fileext = ".gif")
    pixel_morph_animate(imgA, imgB, outfile = outfile, n_frames = 4, show = FALSE)
  }
}
```

---

plot.balance_diagnostics

Plot method for balance diagnostics

Description

Produces a Love plot (dot plot) of standardized differences.

Usage

```
## S3 method for class 'balance_diagnostics'
plot(x, type = c("love", "histogram", "variance"), threshold = 0.1, ...)
```

Arguments

x	A balance_diagnostics object
type	Type of plot: "love" (default), "histogram", or "variance"
threshold	Threshold line for standardized differences (default: 0.1)
...	Additional arguments passed to plotting functions

Value

The balance_diagnostics object (invisibly)

---

plot.matching_result   *Plot method for matching results*

Description

Produces a histogram of pairwise distances from a matching result.

Usage

```
## S3 method for class 'matching_result'
plot(x, type = c("histogram", "density", "ecdf"), ...)
```

Arguments

x	A matching_result object
type	Type of plot: "histogram" (default), "density", or "ecdf"
...	Additional arguments passed to plotting functions

Value

The matching_result object (invisibly)

```
plot.sensitivity_analysis
```
                    Plot method for sensitivity analysis (base graphics)

Description

Plot method for sensitivity analysis (base graphics)

Usage

```
## S3 method for class 'sensitivity_analysis'
plot(x, alpha = 0.05, ...)
```

Arguments

x	A sensitivity_analysis object
alpha	Significance level (default: 0.05)
...	Additional arguments passed to plot

Value

The sensitivity_analysis object (invisibly)

```
preprocess_matching_vars
```
                    Preprocess matching variables with automatic checks and scaling

---

Description

Main preprocessing function that orchestrates variable health checks, categorical encoding, and automatic scaling selection.

Usage

```
preprocess_matching_vars(
  left,
  right,
  vars,
  auto_scale = TRUE,
  scale_method = "auto",
  check_health = TRUE,
  remove_problematic = TRUE,
  verbose = TRUE
)
```

Arguments

`left`	Data frame of left units
`right`	Data frame of right units
`vars`	Character vector of variable names
`auto_scale`	Logical, whether to perform automatic preprocessing (default: TRUE)
`scale_method`	Scaling method: "auto", "standardize", "range", "robust", or FALSE
`check_health`	Logical, whether to check variable health (default: TRUE)
`remove_problematic`	
	Logical, automatically exclude constant/all-NA variables (default: TRUE)
`verbose`	Logical, whether to print warnings (default: TRUE)

Value

A list with class "preprocessing_result" containing:

- `left`: Preprocessed left data frame
- `right`: Preprocessed right data frame
- `vars`: Final variable names (after exclusions)
- `health`: Variable health diagnostics
- `scaling_method`: Selected scaling method
- `excluded_vars`: Variables that were excluded
- `warnings`: List of warnings issued

---

print.balance_diagnostics

Print Method for Balance Diagnostics

Description

Print Method for Balance Diagnostics

Usage

```
## S3 method for class 'balance_diagnostics'
print(x, ...)
```

Arguments

x	A balance_diagnostics object
...	Additional arguments (ignored)

Value

Invisibly returns the input object x.

---

print.distance_object   *Print Method for Distance Objects*

Description

Print Method for Distance Objects

Usage

```
## S3 method for class 'distance_object'
print(x, ...)
```

Arguments

x   A distance_object

...   Additional arguments (ignored)

Value

Invisibly returns the input object x.

---

print.lap_solve_batch_result
                        Print method for batch assignment results

---

Description

Prints a summary and the table of results for a batch of assignment problems solved with lap_solve_batch().

Usage

```
## S3 method for class 'lap_solve_batch_result'
print(x, ...)
```

Arguments

x   A lap_solve_batch_result object.

...   Additional arguments passed to print(). Currently ignored.

Value

Invisibly returns the input object x.

```
print.lap_solve_kbest_result
```
Print method for k-best assignment results

Description

Print method for k-best assignment results

Usage

```
## S3 method for class 'lap_solve_kbest_result'
print(x, ...)
```

Arguments

x   A `lap_solve_kbest_result`.

...   Additional arguments passed to `print()`. Ignored.

Value

Invisibly returns the input object `x`.

```
print.lap_solve_result
```
Print method for assignment results

Description

Nicely prints a `lap_solve_result` object, including the assignments, total cost, and method used.

Usage

```
## S3 method for class 'lap_solve_result'
print(x, ...)
```

Arguments

x   A `lap_solve_result` object.

...   Additional arguments passed to `print()`. Currently ignored.

Value

Invisibly returns the input object `x`.

---

print.matching_result *Print method for matching results*

Description

Print method for matching results

Usage

```
## S3 method for class 'matching_result'
print(x, ...)
```

Arguments

x	A matching_result object
...	Additional arguments (ignored)

Value

Invisibly returns the input object x.

---

print.matchmaker_result

Print method for matchmaker results

Description

Print method for matchmaker results

Usage

```
## S3 method for class 'matchmaker_result'
print(x, ...)
```

Arguments

x	A matchmaker_result object
...	Additional arguments (ignored)

Value

Invisibly returns the input object x.

```
print.preprocessing_result
```
Print method for preprocessing result

Description

Print method for preprocessing result

Usage

```
## S3 method for class 'preprocessing_result'
print(x, ...)
```

Arguments

x   A preprocessing_result object

...   Additional arguments (ignored)

Value

Invisibly returns the input object x.

```
print.sensitivity_analysis
```
Print method for sensitivity analysis

Description

Print method for sensitivity analysis

Usage

```
## S3 method for class 'sensitivity_analysis'
print(x, ...)
```

Arguments

x   A sensitivity_analysis object

...   Additional arguments (ignored)

Value

Invisibly returns the input object x.

---

print.variable_health *Print method for variable health*

Description

Print method for variable health

Usage

```
## S3 method for class 'variable_health'
print(x, ...)
```

Arguments

x	A variable_health object
...	Additional arguments (ignored)

Value

Invisibly returns the input object x.

---

ps_match *Propensity Score Matching*

Description

Matches treated and control units based on estimated propensity scores. Fits a logistic regression model (or accepts a pre-fitted one), computes logit propensity scores, and calls match_couples() with a caliper on the logit scale.

Usage

```
ps_match(
  formula = NULL,
  data = NULL,
  treatment = NULL,
  ps_model = NULL,
  caliper_sd = 0.2,
  method = "auto",
  replace = FALSE,
  ratio = 1L,
  ...
)
```

Arguments

`formula`	Formula for propensity score model (treatment ~ covariates). Required if `ps_model` is NULL.
`data`	Combined dataset containing both treated and control units
`treatment`	Name of the binary treatment column (0/1 or logical)
`ps_model`	Pre-fitted `glm` object (alternative to formula). If provided, `formula` is ignored.
`caliper_sd`	Caliper width in standard deviations of logit(PS). Default: 0.2 (Rosenbaum and Rubin recommendation).
`method`	LAP solver method (default: "auto")
`replace`	If TRUE, match with replacement (default: FALSE)
`ratio`	Integer k for k:1 matching (default: 1)
`...`	Additional arguments passed to [`match_couples()`](#)

Details

The propensity score is the probability of treatment assignment conditional on observed covariates. Matching is performed on the logit of the propensity score (Rosenbaum and Rubin 1985), which provides better distributional properties than matching on the raw probability scale.

The default caliper of 0.2 SD of logit(PS) is recommended by Austin (2011) as removing approximately 98% of bias.

Value

A matching_result object with additional propensity score info in `result$info$ps_model` and `result$info$caliper_value`.

Examples

```
set.seed(42)
n <- 100
data <- data.frame(
  id = seq_len(n),
  treated = rbinom(n, 1, 0.4),
  age = rnorm(n, 50, 10),
  income = rnorm(n, 50000, 15000)
)
result <- ps_match(treated ~ age + income, data = data, treatment = "treated")
print(result)
```

sensitivity_analysis   *Rosenbaum Sensitivity Analysis*

Description

Assesses how sensitive a matched comparison is to hidden bias using Rosenbaum bounds on the Wilcoxon signed-rank statistic.

Usage

```
sensitivity_analysis(
  result,
  left,
  right,
  outcome_var,
  gamma = seq(1, 3, by = 0.25),
  alternative = c("greater", "less", "two.sided"),
  left_id = "id",
  right_id = "id"
)
```

Arguments

result	A matching_result object from [match_couples()](#) or [greedy_couples()](#)
left	Original left (treated) dataset
right	Original right (control) dataset
outcome_var	Name of the outcome column in left and right
gamma	Numeric vector of sensitivity parameters (default: seq(1, 3, by = 0.25)). Gamma = 1 means no hidden bias.
alternative	Direction of the test: "greater" (default), "less", or "two.sided"
left_id	Name of ID column in left (default: "id")
right_id	Name of ID column in right (default: "id")

Details

Rosenbaum (2002, Chapter 4) bounds quantify how much hidden bias (an unobserved confounder) would be needed to explain away the observed treatment effect. The sensitivity parameter Gamma represents the maximum ratio of treatment odds between two matched units:

- Gamma = 1: No hidden bias (standard Wilcoxon test)
- Gamma = 2: One unit could be twice as likely to receive treatment due to an unobserved factor

The function computes upper and lower bounds on the p-value of the Wilcoxon signed-rank test under each level of hidden bias. A finding is "insensitive to bias" if p_upper remains below 0.05 even at large Gamma.

Value

An S3 object of class `sensitivity_analysis` containing:

results Tibble with columns: gamma, t_stat, p_upper, p_lower

n_pairs Number of matched pairs analyzed

critical_gamma Smallest gamma at which p_upper > 0.05

alternative Direction of test

References

Rosenbaum, P.R. (2002). Observational Studies, 2nd edition. Springer.

Examples

```
set.seed(42)
left <- data.frame(id = 1:20, x = rnorm(20), outcome = rnorm(20, 1, 1))
right <- data.frame(id = 21:40, x = rnorm(20), outcome = rnorm(20, 0, 1))
result <- match_couples(left, right, vars = "x")
sens <- sensitivity_analysis(result, left, right, outcome_var = "outcome")
print(sens)
```

---

sinkhorn *'Sinkhorn-Knopp' optimal transport solver*

Description

Compute an entropy-regularized optimal transport plan using the 'Sinkhorn-Knopp' algorithm. Unlike other LAP solvers that return a hard 1-to-1 assignment, this returns a soft assignment (doubly stochastic matrix).

Usage

```
sinkhorn(
  cost,
  lambda = 10,
  tol = 1e-09,
  max_iter = 1000,
  r_weights = NULL,
  c_weights = NULL
)
```

Arguments

cost	Numeric matrix of transport costs. NA or Inf entries are treated as very high cost (effectively forbidden).
lambda	Regularization parameter (default 10). Higher values produce sharper (more deterministic) transport plans; lower values produce smoother distributions. Typical range: 1-100.
tol	Convergence tolerance (default 1e-9).
max_iter	Maximum iterations (default 1000).
r_weights	Optional numeric vector of row marginals (source distribution). Default is uniform. Will be normalized to sum to 1.
c_weights	Optional numeric vector of column marginals (target distribution). Default is uniform. Will be normalized to sum to 1.

Details

The 'Sinkhorn-Knopp' algorithm solves the entropy-regularized optimal transport problem:

$$P^* = \arg\min_P \langle C, P \rangle - \frac{1}{\lambda} H(P)$$

subject to row sums = r_weights and column sums = c_weights.

The entropy term H(P) encourages spread in the transport plan. As lambda -> Inf, the solution approaches the standard (unregularized) optimal transport.

Key differences from standard LAP solvers:

- Returns a soft assignment (probabilities) not a hard 1-to-1 matching
- Supports unequal marginals (weighted distributions)
- Differentiable, making it useful in ML pipelines
- Very fast: O(n^2) per iteration with typically O(1/tol^2) iterations

Use [sinkhorn_to_assignment()](sinkhorn_to_assignment) to round the soft assignment to a hard matching.

Value

A list with elements:

- transport_plan — numeric matrix, the optimal transport plan P. Row sums approximate r_weights, column sums approximate c_weights.
- cost — the transport cost <C, P> (without entropy term).
- u, v — scaling vectors (P = diag(u) * K * diag(v) where K = exp(-lambda*C)).
- converged — logical, whether the algorithm converged.
- iterations — number of iterations used.
- lambda — the regularization parameter used.

References

Cuturi, M. (2013). 'Sinkhorn Distances': Lightspeed Computation of Optimal Transport. *Advances in Neural Information Processing Systems*, 26.

See Also

[assignment()](#) for hard 1-to-1 matching, [sinkhorn_to_assignment()](#) to round soft assignments.

Examples

```
cost <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, byrow = TRUE)

# Soft assignment with default parameters
result <- sinkhorn(cost)
print(round(result$transport_plan, 3))

# Sharper assignment (higher lambda)
result_sharp <- sinkhorn(cost, lambda = 50)
print(round(result_sharp$transport_plan, 3))

# With custom marginals (more mass from row 1)
result_weighted <- sinkhorn(cost, r_weights = c(0.5, 0.25, 0.25))
print(round(result_weighted$transport_plan, 3))

# Round to hard assignment
hard_match <- sinkhorn_to_assignment(result)
print(hard_match)
```

---

sinkhorn_to_assignment

Round 'Sinkhorn' transport plan to hard assignment

Description

Convert a soft transport plan from [sinkhorn()](#) to a hard 1-to-1 assignment using greedy rounding.

Usage

```
sinkhorn_to_assignment(result)
```

Arguments

result   Either a result from [sinkhorn()](#) or a transport plan matrix.

Details

Greedy rounding iteratively assigns each row to its most probable column, ensuring no column is assigned twice. This may not give the globally optimal hard assignment; for that, use the transport plan as a cost matrix with [assignment()](#).

Value

Integer vector of column assignments (1-based), same format as assignment().

See Also

sinkhorn()

Examples

```
cost <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, byrow = TRUE)
result <- sinkhorn(cost, lambda = 20)
hard_match <- sinkhorn_to_assignment(result)
print(hard_match)
```

---

summary.balance_diagnostics

Summary method for balance diagnostics

Description

Summary method for balance diagnostics

Usage

```
## S3 method for class 'balance_diagnostics'
summary(object, ...)
```

Arguments

object	A balance_diagnostics object
...	Additional arguments (ignored)

Value

A list containing summary statistics (invisibly)

summary.distance_object

Summary Method for Distance Objects

Description

Summary Method for Distance Objects

Usage

```
## S3 method for class 'distance_object'
summary(object, ...)
```

Arguments

object	A distance_object
...	Additional arguments (ignored)

Value

Invisibly returns the input object.

summary.lap_solve_kbest_result

Get summary of k-best results

Description

Extract summary information from k-best assignment results.

Usage

```
## S3 method for class 'lap_solve_kbest_result'
summary(object, ...)
```

Arguments

object	An object of class lap_solve_kbest_result.
...	Additional arguments (unused).

Value

A tibble with one row per solution containing:

- rank: solution rank
- solution_id: solution identifier
- total_cost: total cost of the solution
- n_assignments: number of assignments in the solution

---

summary.matching_result

Summary method for matching results

Description

Summary method for matching results

Usage

```
## S3 method for class 'matching_result'
summary(object, ...)
```

Arguments

object	A matching_result object
...	Additional arguments (ignored)

Value

A list containing summary statistics (invisibly)

---

summary.sensitivity_analysis

Summary method for sensitivity analysis

Description

Summary method for sensitivity analysis

Usage

```
## S3 method for class 'sensitivity_analysis'
summary(object, ...)
```

Arguments

`object`	A sensitivity_analysis object
`...`	Additional arguments (ignored)

Value

A list containing summary statistics (invisibly)

`update_constraints` *Update Constraints on Distance Object*

Description

Apply new constraints to a precomputed distance object without recomputing the underlying distances. This is useful for exploring different constraint scenarios quickly.

Usage

```
update_constraints(dist_obj, max_distance = Inf, calipers = NULL)
```

Arguments

`dist_obj`	A distance_object from `compute_distances()`
`max_distance`	Maximum allowed distance (pairs with distance > max_distance become Inf)
`calipers`	Named list of per-variable calipers

Details

This function creates a new distance_object with modified constraints applied to the cost matrix. The original distance_object is not modified.

Constraints:

- `max_distance`: Sets cost to Inf for pairs exceeding this threshold
- `calipers`: Per-variable restrictions (e.g., calipers = list(age = 5))

The function returns a new object rather than modifying in place, following R's copy-on-modify semantics.

Value

A new distance_object with updated cost_matrix

Examples

```
left <- data.frame(id = 1:5, age = c(25, 30, 35, 40, 45))
right <- data.frame(id = 6:10, age = c(24, 29, 36, 41, 44))
dist_obj <- compute_distances(left, right, vars = "age")

# Apply constraints
constrained <- update_constraints(dist_obj, max_distance = 2)
result <- match_couples(constrained)
```

Index