Package 'dgraphs'

August 20, 2026

Title Data-Derived Graph Construction Utilities

Version 0.1.0

Description Constructs data-derived graphs from numerical observations using mutual, shared-neighbor, intersection, geodesic, radius, adaptive-radius, and minimum-spanning-tree completion methods. Provides graph conversion, pruning, diagnostics, spectral embedding, endpoint detection, and path utilities. The implemented graph constructions include methods described by Jarvis and Patrick (1973) <doi:10.1109/T-C.1973.223640>, Brito et al. (1997) <doi:10.1016/S0167-7152(96)00213-1>, Berry and Sauer (2019) <doi:10.3934/fods.2019001>, and Gower and Ross (1969) <doi:10.2307/2346439>.

License MIT + file LICENSE

Copyright file inst/COPYRIGHTS

URL https://github.com/pgajer/dgraphs

BugReports https://github.com/pgajer/dgraphs/issues

Encoding UTF-8

Language en-US

SystemRequirements C++17

Depends R (>= 4.1.0)

Imports FNN, igraph (>= 2.2.0), parallel

LinkingTo Rcpp, RcppEigen

Suggests Matrix, testthat (>= 3.0.0)

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

NeedsCompilation yes

Author Peter Gajer [aut, cre],
Sunil Arya [ctb] (ANN library),
David M. Mount [ctb] (ANN library),
University of Maryland [cph] (ANN library),
Yixuan Qiu [ctb, cph] (Spectra library),

Anna Araslanova [ctb, cph] (Spectra LOBPCG solver),
Gael Guennebaud [ctb, cph] (Spectra linear algebra routines),
Jitse Niesen [ctb, cph] (Spectra linear algebra routines),
Netherlands eScience Center [ctb, cph] (Spectra eigensolver routines)

Maintainer Peter Gajer <pgajer@gmail.com>

Repository CRAN

Date/Publication 2026-08-20 17:00:11 UTC

Contents

`adjlist.to.igraph` *Convert an Adjacency List to igraph*

Description

Convert an Adjacency List to igraph

Usage

```
adjlist.to.igraph(adj.list)
```

Arguments

adj.list	A 1-based adjacency list.

Value

An undirected igraph graph.

---

as_igraph *Convert a Basin Graph to igraph*

Description

Convert a Basin Graph to igraph

Usage

```
as_igraph(gflow.graph, include.vertex.attrs = TRUE, include.edge.attrs = TRUE)
```

Arguments

gflow.graph	A basin graph list with adjacency, weight, intersection, and basin metadata fields. The argument name is retained for backward compatibility.
include.vertex.attrs	Logical; include basin metadata as vertex attributes.
include.edge.attrs	Logical; include edge weights and intersection sizes as edge attributes.

Value

An igraph object.

`build.iknn.graphs.and.selectk` *Build iKNN Graphs and Select a Neighborhood Size*

Description

Builds an iKNN graph sequence and selects k from structural edit-distance stability, label-mixing stability, or both.

Usage

```
build.iknn.graphs.and.selectk(
  X,
  kmin,
  kmax,
  method = c("both", "edit", "mixing", "none"),
  pca.dim = 100,
  variance.explained = 0.99,
  trim.disconnected = TRUE,
  edit.min.lcc.frac = 1,
  edit.eps = 0.05,
  labels = NULL,
  perm.blocks = NULL,
  mixing.metric = c("homophily.effect", "homophily.z", "homophily.adjusted",
    "assortativity.effect", "assortativity.z", "homophily", "assortativity",
    "conductance.median", "conductance.wmean"),
  mixing.min.lcc.frac = 0.98,
  mixing.eps = 0.05,
  mixing.require.local.extremum = TRUE,
  mixing.window = 1L,
  n.perm = 200L,
  use.edge.weights = TRUE,
  weights.are.edge.lengths = FALSE,
  affinity.method = c("exp", "inv"),
  affinity.sigma = NULL,
  affinity.sigma.from = c("k.cc.mixing", "k.max.lcc", "k.trim"),
  affinity.eps = 1e-08,
  simplify.multiple = TRUE,
  seed = 1L,
  n.cores = 1L,
  verbose = TRUE,
  ...
)
```

Arguments

`X`	Numeric observation-by-feature matrix.

`kmin`, `kmax` Minimum and maximum neighborhood sizes.

`method` Selection criterion.

`pca.dim`, `variance.explained`
PCA controls forwarded to `create.iknn.graphs()`.

`trim.disconnected`
Logical; trim to a largest connected component when the requested connectivity tail is absent.

`edit.min.lcc.frac`, `edit.eps`
Connectivity and tolerance controls for edit-distance selection.

`labels`, `perm.blocks`
Optional labels and permutation blocks for mixing selection.

`mixing.metric`, `mixing.min.lcc.frac`, `mixing.eps`
Mixing criterion, connectivity threshold, and tolerance.

`mixing.require.local.extremum`, `mixing.window`
Local-extremum controls for the mixing curve.

`n.perm` Number of label permutations.

`use.edge.weights`, `weights.are.edge.lengths`
Edge-weight interpretation.

`affinity.method`, `affinity.sigma`, `affinity.sigma.from`, `affinity.eps`
Controls for converting edge lengths to affinities.

`simplify.multiple`
Logical; simplify loops and multiple edges.

`seed` Random seed.

`n.cores` Number of worker processes.

`verbose` Logical; report progress.

`...` Additional arguments forwarded to `create.iknn.graphs()`.

Value

An object of class `"build_iknn_graphs_and_selectk"` containing the graph sequence, connectivity diagnostics, selection curves, selected neighborhood sizes, trimming metadata, and call parameters.

`calculate.edit.distances` *Calculate Edit Distances Between Sequential Graphs*

Description

Calculates edit distances between pairs of graphs separated by a specified offset in the sequence of k values.

Usage

```
calculate.edit.distances(
  k.values,
  graph.data,
  offset = 1,
  edge.cost = 1,
  weight.cost.factor = 0.1,
  verbose = FALSE
)
```

Arguments

k.values	Numeric vector of k values corresponding to the loaded graphs.
graph.data	List containing graph data as produced by load.graph.data.
offset	Positive integer specifying the offset between compared graphs (default: 1). An offset of 1 compares consecutive graphs.
edge.cost	Cost parameter passed to graph.edit.distance (default: 1).
weight.cost.factor	Weight cost factor passed to graph.edit.distance (default: 0.1).
verbose	Logical indicating whether to show progress messages (default: FALSE).

Details

For a sequence of k values and an offset of 1, this function compares:

- Graph at k[1] with graph at k[2]
- Graph at k[2] with graph at k[3]
- And so on...

Value

A list containing:

indices	Integer vector of indices in the k.values sequence
k.values	Numeric vector of k values corresponding to the first graph in each comparison
distances	Numeric vector of calculated edit distances

Examples

```
graph.data <- list(
  adj.list = list(
    `5` = list(c(2), c(1, 3), c(2)),
    `10` = list(c(2, 3), c(1, 3), c(1, 2)),
    `15` = list(c(2, 3), c(1), c(1))
  ),
  dist.list = list(
    `5` = list(1, c(1, 2), 2),
    `10` = list(c(1, 2), c(1, 3), c(2, 3)),
```

```
    `15` = list(c(1, 1.5), 1, 1.5)
  )
)
k.vals <- c(5, 10, 15)
distances <- calculate.edit.distances(k.vals, graph.data, offset = 1)
distances$distances
```

`compare.adj.lists` *Compare Two Adjacency Lists*

Description

Compare Two Adjacency Lists

Usage

```
compare.adj.lists(adj.list1, adj.list2)
```

Arguments

`adj.list1`	First adjacency list.
`adj.list2`	Second adjacency list.

Value

`TRUE` when each vertex has the same neighbor set in both lists, ignoring order; otherwise `FALSE`.

`compare.paths` *Compare Paths Across Hop Limits*

Description

Compare Paths Across Hop Limits

Usage

```
compare.paths(x, from, to)
```

Arguments

`x`	A `path.graph.series` object.
`from`	Source vertex index.
`to`	Target vertex index.

Value

Data frame describing path availability and length by hop limit.

`compute.geodesic.stats` *Compute Geodesic Statistics for Grid Vertices*

Description

Analyzes how the number of geodesics scales with radius for each grid vertex. This function helps determine if collapsing/clustering geodesics is necessary and provides insights into the graph structure around grid vertices.

Usage

```
compute.geodesic.stats(
  adj.list,
  weight.list,
  min.radius = 0.2,
  max.radius = 0.5,
  n.steps = 5,
  n.packing.vertices = length(adj.list),
  max.packing.iterations = 20,
  packing.precision = 1e-04,
  verbose = FALSE
)
```

Arguments

`adj.list`	List of integer vectors. Each vector contains indices of vertices adjacent to the corresponding vertex. Indices should be 1-based.
`weight.list`	List of numeric vectors. Each vector contains weights of edges corresponding to adjacencies in adj.list.
`min.radius`	Numeric. Minimum radius as a fraction of graph diameter.
`max.radius`	Numeric. Maximum radius as a fraction of graph diameter.
`n.steps`	Integer. Number of radius steps to test.
`n.packing.vertices`	Integer. Number of vertices in the grid/packing.
`max.packing.iterations`	Integer. Maximum iterations for packing algorithm.
`packing.precision`	Numeric. Precision parameter for packing algorithm.
`verbose`	Logical. Whether to print progress information.

Value

A list of class "geodesic_stats" containing:

radii Numeric vector. Actual radii used in the analysis.

geodesic_rays Matrix. Number of geodesic rays for each vertex at each radius.

composite_geodesics Matrix. Number of composite geodesics for each vertex at each radius.

path_overlap List of matrices. Overlap statistics for each vertex at each radius.

grid_vertices Integer vector. The vertices selected as grid vertices.

summary Data frame. Summary statistics for each radius.

`compute.graph.diameter`

Compute a Weighted Graph Diameter

Description

Compute a Weighted Graph Diameter

Usage

```
compute.graph.diameter(adj.list, weight.list)
```

Arguments

`adj.list`	Adjacency list.
`weight.list`	Edge-length list matching `adj.list`.

Value

A list containing the diameter and the farthest path details.

`compute.graph.distance`

Compute a Weighted Shortest-Path Distance

Description

Compute a Weighted Shortest-Path Distance

Usage

```
compute.graph.distance(
  star.obj = NULL,
  i,
  j,
  adj.list = NULL,
  edge.lengths = NULL
)
```

Arguments

`star.obj`	Optional object containing `adj.list` and `edge.lengths`.
`i`	Source vertex.
`j`	Target vertex.
`adj.list`	Adjacency list.
`edge.lengths`	Edge-length list matching `adj.list`.

Value

Numeric shortest-path distance.

`compute.graph.endpoint.scores` *Compute Graph Endpoint Scores from a 3D Embedding*

Description

Computes endpointness diagnostics for graph vertices by selecting graph-geodesic neighborhoods and evaluating how concentrated the corresponding 3D embedding directions are. This targets terminal arm tips in an embedding rather than graph-theoretic degree-1 vertices.

For each vertex x, let $u_1, \ldots, u_m$ be unit vectors in the embedding from x to graph-geodesic neighbors of x. The function computes:

$$s_{\min}(x) = \min_{i<j} u_i^\top u_j$$

$$s_q(x) = \text{quantile}\{u_i^\top u_j : i < j\}$$

$$m(x) = \left\| \frac{1}{\sum_i w_i} \sum_i w_i u_i \right\|$$

$$\text{score}(x) = m(x)\,\frac{1 + s_q(x)}{2}$$

Neighborhoods are selected in graph-geodesic space while directions are taken from the supplied 3D embedding.

Usage

```
compute.graph.endpoint.scores(
  adj.list,
  weight.list,
  layout.3d,
  neighborhood = c("geodesic_k", "geodesic_radius"),
  k = c(10L, 20L, 30L),
  radius = NULL,
  q = 0.1,
  neighbor.weighting = c("uniform", "inverse_distance", "gaussian"),
  gaussian.sigma = NULL,
  min.neighborhood.size = 3L,
  scale.aggregate = c("mean", "median", "min", "max"),
  verbose = FALSE
)
```

Arguments

`adj.list`	Graph adjacency list (1-based vertex indices).
`weight.list`	Edge-length list aligned with `adj.list`.
`layout.3d`	Numeric matrix or data frame with 3 columns giving the 3D embedding coordinates for graph vertices.
`neighborhood`	Character string. Either `"geodesic_k"` (default) or `"geodesic_radius"`.
`k`	Integer vector of geodesic neighborhood sizes used when `neighborhood = "geodesic_k"`.
`radius`	Positive numeric vector of geodesic radii used when `neighborhood = "geodesic_radius"`.
`q`	Quantile used for `s.q`. Must lie in `[0, 1]`. Default is `0.1`.
`neighbor.weighting`	Character string controlling neighbor weights for `m` and weighted pair quantiles. One of `"uniform"` (default), `"inverse_distance"`, or `"gaussian"`.
`gaussian.sigma`	Optional positive numeric scale used when `neighbor.weighting = "gaussian"`. If `NULL`, the median positive geodesic neighborhood distance is used per vertex and scale.
`min.neighborhood.size`	Minimum number of usable neighbors required after removing zero-length embedding vectors. Must be at least `2`.
`scale.aggregate`	How to aggregate scores across multiple neighborhood scales. One of `"mean"` (default), `"median"`, `"min"`, or `"max"`.
`verbose`	Logical; if `TRUE`, prints progress.

Value

A list with aggregated per-vertex scores, per-scale score matrices, and neighborhood diagnostics.

`compute.graph.summary.pmf` *Compute a graph summary as a probability mass function*

Description

Converts a graph-like object into a named probability mass function (PMF) describing one selected graph summary. This provides a standardized bridge between graph objects and divergence functions such as `jensen.shannon.divergence()`.

Supported v1 summaries are:

- `"degree_distribution"`
- `"edge_weight_distribution"`
- `"component_size_distribution"`
- `"neighborhood_label_distribution"`

`"component_size_distribution"` is included primarily as a diagnostic for disconnectedness / fragmentation, not as a general-purpose default stability summary.

Usage

```
compute.graph.summary.pmf(
  graph,
  summary = c("degree_distribution", "edge_weight_distribution",
    "component_size_distribution", "neighborhood_label_distribution"),
  labels = NULL,
  weight.list = NULL,
  bins = NULL,
  n.bins = 20L,
  bin.method = c("auto", "fixed_width", "fixed_quantile", "explicit"),
  normalize = TRUE,
  support = NULL,
  zero.pad = TRUE,
  simplify.multiple = TRUE,
  directed = FALSE,
  return.details = TRUE
)
```

Arguments

`graph`	Graph-like object. Supported inputs are: • an object with `adj_list` and optional `weight_list` • an object with `pruned_adj_list` and optional `pruned_weight_list` • a plain adjacency list (list of integer vectors)
`summary`	Summary type to compute.

labels	Optional label vector used for "neighborhood_label_distribution". Must have length equal to the number of vertices.
weight.list	Optional edge-weight list used when graph is a plain adjacency list.
bins	Optional numeric vector of histogram breaks used for "edge_weight_distribution".
n.bins	Integer number of bins used when bins is NULL and the summary is "edge_weight_distribution".
bin.method	Character string controlling automatic bin construction for "edge_weight_distribution".
normalize	Logical; if TRUE, convert counts to a PMF.
support	Optional support to align the resulting PMF to.
zero.pad	Logical; if TRUE and support is provided, missing support entries are padded with zero mass.
simplify.multiple	Logical; if TRUE, duplicate edges are collapsed before computing summaries.
directed	Logical; if FALSE (default), treat edges as undirected.
return.details	Logical; if TRUE, return a structured list. Otherwise return the named PMF directly.

Value

If return.details = TRUE, a list with fields:

- summary: summary type used.
- pmf: named probability mass function.
- support: support of the PMF.
- counts: named raw-count vector before normalization.
- metadata: list of summary-specific metadata.

Otherwise, returns the named PMF vector.

---

compute.graph.summary.stability

Compute graph-summary stability across a graph sequence

Description

Computes a divergence-based stability curve across consecutive graphs in a sequence. This function provides a generalized sequence-level analogue of the degree-profile Jensen-Shannon calculation currently used in compute.stability.metrics().

Usage

```
compute.graph.summary.stability(
  graphs,
  summary = c("degree_distribution", "edge_weight_distribution",
    "component_size_distribution", "neighborhood_label_distribution"),
  divergence = c("js"),
  labels = NULL,
  summary.args = list(),
  k.values = NULL,
  graph.type = c("geom", "isize"),
  return.details = TRUE
)
```

Arguments

graphs	Either a list of graph-like objects or an object of class "iknn_graphs".
summary	Summary type to compare across consecutive graph pairs.
divergence	Divergence type. Currently only "js".
labels	Optional shared vertex-label vector used for "neighborhood_label_distribution".
summary.args	Optional named list forwarded to graph.summary.divergence().
k.values	Optional integer vector of k values. If graphs is an "iknn_graphs" object and k.values is NULL, the values are derived from its kmin and kmax attributes.
graph.type	If graphs is an "iknn_graphs" object, choose either "geom" or "isize".
return.details	Logical; if TRUE, return a structured list.

Value

If return.details = TRUE, a list with fields:

- summary: summary type used.
- divergence: divergence type used.
- k.values: full k sequence.
- k.transition.values: k values corresponding to consecutive-graph transitions.
- values: vector of divergence values of length length(k.values) - 1.
- details: optional pairwise details for each transition.

Otherwise, returns only the divergence vector.

compute.stability.metrics

Compute Stability Metrics Across a Sequence of IkNN Graphs

Description

Computes stability diagnostics across the pruned IkNN graphs produced by `create.iknn.graphs()`. Metrics include edit distances between consecutive graphs, graph-summary Jensen-Shannon divergence, edge-count diagnostics, local minima, and smoothed trend fits for plotting.

Usage

```
compute.stability.metrics(
  graphs,
  graph.type = c("geom", "isize"),
  summary = c("degree_distribution", "edge_weight_distribution",
    "component_size_distribution", "neighborhood_label_distribution"),
  divergence = c("js"),
  labels = NULL,
  summary.args = list()
)
```

Arguments

`graphs`	An object of class `"iknn_graphs"` returned by `create.iknn.graphs()`.
`graph.type`	Character string, either `"geom"` or `"isize"`.
`summary`	Graph-summary family used for the divergence curve.
`divergence`	Divergence used for consecutive-graph comparison. Currently only `"js"`.
`labels`	Optional vertex-label vector used when `summary = "neighborhood_label_distribution"`.
`summary.args`	Optional named list forwarded to `compute.graph.summary.stability()`.

Value

An object of class `"iknn_stability_metrics"`.

compute.vertex.geodesic.stats

Compute Geodesic Statistics for a Single Grid Vertex

Description

Analyzes how the number of geodesics scales with radius for a specific grid vertex.

Usage

```
compute.vertex.geodesic.stats(
  adj.list,
  weight.list,
  grid.vertex,
  min.radius = 0.2,
  max.radius = 0.5,
  n.steps = 5,
  n.packing.vertices = length(adj.list),
  packing.precision = 1e-04
)
```

Arguments

`adj.list`	List of integer vectors. Each vector contains indices of vertices adjacent to the corresponding vertex. Indices should be 1-based.
`weight.list`	List of numeric vectors. Each vector contains weights of edges corresponding to adjacencies in adj.list.
`grid.vertex`	Integer. The grid vertex to analyze.
`min.radius`	Numeric. Minimum radius as a fraction of graph diameter.
`max.radius`	Numeric. Maximum radius as a fraction of graph diameter.
`n.steps`	Integer. Number of radius steps to test.
`n.packing.vertices`	Integer. Number of vertices in the grid/packing.
`packing.precision`	Numeric. Precision threshold for packing convergence.

Value

A data frame of class "vertex_geodesic_stats" with columns:

radius Actual radius used.

rays Number of geodesic rays within radius.

composite_geodesics Number of composite geodesics.

overlap_min Minimum overlap ratio.

overlap_p05 5th percentile overlap ratio.

overlap_p25 25th percentile overlap ratio.

overlap_median Median overlap ratio.

overlap_p75 75th percentile overlap ratio.

overlap_p95 95th percentile overlap ratio.

overlap_max Maximum overlap ratio.

convert.adjacency.list.to.adjacency.matrix *Convert an Adjacency List to an Adjacency Matrix*

Description

Convert an Adjacency List to an Adjacency Matrix

Usage

```
convert.adjacency.list.to.adjacency.matrix(
  adj.list,
  weight.list = NULL,
  mode = "undirected",
  remove.self.loops = TRUE
)
```

Arguments

adj.list	Adjacency list.
weight.list	Optional edge-weight list aligned with adj.list.
mode	Either "undirected" or "directed".
remove.self.loops	Logical; if TRUE, remove diagonal entries.

Value

Numeric adjacency matrix.

convert.adjacency.to.edge.matrix *Convert an Adjacency List to an Edge Matrix*

Description

Convert an Adjacency List to an Edge Matrix

Usage

```
convert.adjacency.to.edge.matrix(adj.list, weights.list = NULL)
```

Arguments

adj.list	A 1-based adjacency list.
weights.list	Optional edge-weight list aligned with adj.list.

Value

A list with `edge.matrix` and `weights`.

`convert.to.undirected` *Convert a Directed Adjacency List to an Undirected Adjacency List*

Description

Convert a Directed Adjacency List to an Undirected Adjacency List

Usage

```
convert.to.undirected(adj.list)
```

Arguments

`adj.list`	Directed adjacency list.

Value

An adjacency list with reciprocal edges added and duplicate neighbors removed.

`convert.weighted.adjacency.matrix.to.adjacency.list` *Convert a Weighted Adjacency Matrix to Lists*

Description

Convert a Weighted Adjacency Matrix to Lists

Usage

```
convert.weighted.adjacency.matrix.to.adjacency.list(A)
```

Arguments

`A`	Numeric square adjacency matrix.

Value

A list with `adjacency.list` and `weights.list`.

count.edges *Count Edges in an Undirected Adjacency List*

Description

Count Edges in an Undirected Adjacency List

Usage

```
count.edges(adj.list)
```

Arguments

adj.list	Adjacency list for an undirected graph.

Value

Number of undirected edges.

cpp.create.rknn.graphs *Compute Adaptive Radius-kNN Graphs With Batched ANN Search*

Description

Tentative C++-backed counterpart to `create.rknn.graphs()`. It builds one ANN kd-tree, computes nearest-neighbor distances through the maximal requested k value once, derives each requested `k.scale` from that shared result, and materializes adaptive-radius edge tables for all k values in C++. The existing R finalization path is then used for pruning, lifecycle branches, and optional component repair.

This function is exported as a diagnostic helper while the native backend is being benchmarked and validated. Ordinary callers should use `create.rknn.graphs()` with the default `backend = "auto"` or with `backend = "cpp"` to request this native backend explicitly.

Usage

```
cpp.create.rknn.graphs(X, kmin = NULL, kmax = NULL, ..., k.values = NULL)
```

Arguments

X	Numeric matrix or data frame with observations in rows.
kmin, kmax	Optional integer scalars defining the inclusive k range. Required when k.values is NULL.
...	Additional arguments forwarded to create.rknn.graph() with type = "adaptive.radius". The arguments type, k.scale, and radius are reserved by this plural constructor.
k.values	Optional integer vector of k values to evaluate. When supplied, k.values is used instead of kmin:kmax, and the returned graph order follows the supplied vector.

Value

A "rknn_graphs" object with the same default structure as create.rknn.graphs(). When return.timing = TRUE, timing is attached at the graph-sequence level because the ANN setup and max-k scale search are shared across k values.

See Also

create.rknn.graphs()

Examples

```
X <- matrix(c(0, 1, 3, 4), ncol = 1)
result <- cpp.create.rknn.graphs(
  X,
  k.values = c(1, 2),
  graph.detail = "minimal",
  prune.method = "none"
)
names(result$graphs)
```

---

create.bi.kNN.chain.graph

Create a Bi-kNN Chain Graph

Description

Create a Bi-kNN Chain Graph

Usage

```
create.bi.kNN.chain.graph(n.vertices = 5, k = 1, x = NULL, y = NULL)
```

Arguments

n.vertices	Number of vertices, ignored when x is supplied.
k	Number of neighbors on each side of a vertex in sorted order.
x	Optional numeric coordinate used to order vertices.
y	Optional response vector sorted along with x.

Value

A list with adj.list, edge.lengths, x.sorted, and y.sorted.

---

create.bipartite.graph *Create a Bipartite Graph*

Description

Create a Bipartite Graph

Usage

```
create.bipartite.graph(n1, n2)
```

Arguments

n1	Number of vertices in the first part.
n2	Number of vertices in the second part.

Value

An adjacency list for the complete bipartite graph.

---

create.chain.graph *Create a Chain Graph*

Description

Create a Chain Graph

Usage

```
create.chain.graph(n.vertices = NULL, x = NULL, y = NULL)
```

Arguments

n.vertices	Number of vertices. Required when x is NULL.
x	Optional coordinate vector; if supplied, vertices are ordered by x.
y	Optional response vector sorted along with x.

Value

A list with adj.list, edge.lengths, x.sorted, and y.sorted.

---

create.chain.graph.with.offset *Create a Chain Graph with Offset Vertex Labels*

Description

Create a Chain Graph with Offset Vertex Labels

Usage

```
create.chain.graph.with.offset(n, offset = 0)
```

Arguments

n	The number of vertices in the graph.
offset	An offset in indexing the vertices of the graph.

Value

A chain graph adjacency list.

---

create.circular.graph *Create a Circular Graph*

Description

Create a Circular Graph

Usage

```
create.circular.graph(n)
```

Arguments

n	Number of vertices.

Value

An undirected cycle graph adjacency list.

`create.cknn.graph` *Compute a Continuous-kNN Graph*

Description

Creates an undirected continuous-kNN graph using local scale distances. Let σ_i be the distance from observation i to its `k.scale`-th non-self nearest neighbor. Vertices are adjacent when

$$\|x_i - x_j\|_2 \leq \delta\sqrt{\sigma_i \sigma_j}.$$

This is a convenience wrapper around `create.rknn.graph()` with `type = "adaptive.radius"` and `radius.rule = "geomean"`. It is useful as a named data-derived graph construction in geodesic reconstruction benchmarks.

Usage

```
create.cknn.graph(
  X,
  k.scale,
  delta = 1,
  radius.search = c("ann", "all.pairs"),
  return.timing = FALSE,
  graph.detail = c("full", "minimal"),
  prune.method = c("none", "local.geodesic", "global.geodesic.ratio"),
  max.path.edge.ratio.deviation.thld = 0.1,
  path.edge.ratio.percentile = 0.5,
  prune.tau = 1.05,
  prune.local.k = NULL,
  with.pruned.edge.stats = FALSE,
  connect.components = FALSE,
  connect.method = c("component.mst", "component.mst.ann", "global.mst"),
  bridge.k = NULL,
  bridge.k.max = NULL,
  bridge.growth = 2
)
```

Arguments

`X`	Numeric matrix or data frame with observations in rows.
`k.scale`	Positive integer smaller than `nrow(X)`. Defines local scale distances σ_i.
`delta`	Positive numeric scalar multiplying the geometric-mean adaptive radius.
`radius.search`	Character scalar. `"ann"` uses the bundled ANN kd-tree exact radius-search backend inherited from `create.rknn.graph()`. `"all.pairs"` uses the direct $O(n^2)$ reference path.

`return.timing` Logical scalar. If `TRUE`, attach the same construction timing table returned by `create.rknn.graph()`, including ANN setup, local-scale search, fixed-radius candidate search, edge materialization, and graph finalization when `radius.search = "ann"`.

`graph.detail` Character scalar. `"full"` returns the complete graph lifecycle object. `"minimal"` returns only core graph fields and is allowed only with `prune.method = "none"` and `connect.components = FALSE`.

`prune.method` Character scalar. `"none"` disables geometric pruning. `"local.geodesic"` applies the experimental local geometric pruning stage before optional MST connectivity repair. `"global.geodesic.ratio"` applies whole-graph geodesic-ratio pruning before optional MST connectivity repair.

`max.path.edge.ratio.deviation.thld`
Numeric scalar in `[0, 0.2)`. For `prune.method = "global.geodesic.ratio"`, an edge may be removed when the shortest alternative path is at most `1 + max.path.edge.ratio.deviat` times the direct edge length.

`path.edge.ratio.percentile`
Numeric scalar in `[0, 1]`. For `prune.method = "global.geodesic.ratio"`, only edges at or above this edge length percentile are considered.

`prune.tau` Numeric scalar greater than 1. For local geometric pruning, an edge may be removed when a retained local alternative path is at most this multiplicative factor times the direct edge length.

`prune.local.k` Integer scalar or `NULL`. Number of nearest neighbors used to form local neighborhoods for pruning. For fixed-radius graphs, `NULL` uses the median positive raw graph degree, clipped to `[1, n - 1]`. For adaptive-radius graphs, `NULL` defaults to `k.scale`.

`with.pruned.edge.stats`
Logical scalar. If `TRUE`, return a data frame with one row per locally pruned edge.

`connect.components`
Logical scalar. If `TRUE`, add MST bridge edges so the final graph is connected whenever possible.

`connect.method` Character scalar. `"component.mst"` adds exact shortest inter-component bridges. `"component.mst.ann"` tries sparse ANN bridge candidates before automatic exact fallback. `"global.mst"` unions the graph with the full Euclidean MST.

`bridge.k` Integer scalar or `NULL`. Initial ANN bridge neighborhood size for `connect.method = "component.mst.ann"`.

`bridge.k.max` Integer scalar or `NULL`. Maximum ANN bridge neighborhood size before exact fallback.

`bridge.growth` Numeric scalar greater than 1. Multiplicative growth factor for ANN bridge neighborhoods.

Value

A list inheriting from `"cknn_graph"` and `"adaptive_radius_graph"`. It contains the same lifecycle fields as `create.rknn.graph()` with `type = "adaptive.radius"`, `radius_rule = "geomean"`, `radius_factor = delta`, and `graph_rule = "continuous.knn"`.

Examples

```
X <- matrix(c(0, 1, 3), ncol = 1)
create.cknn.graph(X, k.scale = 1, delta = 1)$edge_matrix
```

`create.cmst.graph` *Construct a Minimal Spanning Tree (MST) Completion Graph*

Description

Constructs a completion of the Minimal Spanning Tree (MST) graph from a numeric data matrix. The graph is constructed in two steps: first, a MST is built connecting all data points using minimal total edge length; then, additional edges are added between any pair of points whose Euclidean distance is less than or equal to a quantile threshold based on the MST edge length distribution.

Usage

```
create.cmst.graph(
  X,
  q.thld = 0.9,
  pca.dim = 100,
  variance.explained = 0.99,
  verbose = TRUE
)
```

Arguments

`X`	A numeric matrix or data frame of shape $n \times d$, where each row represents a data point in d-dimensional space.
`q.thld`	Numeric scalar between 0 and 1 (exclusive). The quantile threshold for MST completion. Edges are added between points whose distance is less than or equal to the q.thld-quantile of MST edge weights. Default is 0.9.
`pca.dim`	Positive integer or NULL. If provided and `ncol(X)` > `pca.dim`, dimensionality is reduced to this many principal components before graph construction. Must be less than min(n-1, p) where n is the number of observations and p is the number of variables. Default is 100.
`variance.explained`	Numeric between 0 and 1 (exclusive) or NULL. If provided, selects the minimal number of PCA components such that this proportion of total variance is retained (up to `pca.dim` components). Ignored if `pca.dim` is NULL. Default is 0.99.
`verbose`	Logical. If `TRUE`, prints progress messages during computation. Default is `TRUE`.

Details

The MST completion process enhances connectivity in the original MST by adding edges between vertices that are "close" according to the distance distribution in the MST. This can be useful for creating more robust graph representations of data that maintain local structure while avoiding long-range connections.

When PCA is applied (either through `pca.dim` or `variance.explained`), the data is first centered and projected onto the leading principal components before graph construction. This can significantly reduce computation time for high-dimensional data while preserving the most important variation.

Value

An object of class `mst_completion_graph`, which is a list containing:

`mst_adj_list` A list of length n, where element i contains the indices of vertices adjacent to vertex i in the MST.

`mst_weight_list` A list of length n, where element i contains the edge weights corresponding to the adjacent vertices in `mst_adj_list[[i]]`.

`cmst_adj_list` A list of length n containing adjacency information for the completed MST graph.

`cmst_weight_list` A list of length n containing edge weights for the completed MST graph.

`mst_edge_weights` Numeric vector containing all unique MST edge weights.

`cmst_distance_threshold` Numeric scalar containing the actual Euclidean distance threshold used for completion.

The returned object also has the following attributes:

`q_thld` The quantile threshold used for MST completion.

`pca` If PCA was applied, a list containing:

- `original_dim`: The original number of dimensions
- `n_components`: The number of PCA components used
- `variance_explained`: The proportion of variance explained
- `cumulative_variance`: Cumulative variance by component (if applicable)

`call` The matched function call.

References

Gower, J. C., & Ross, G. J. S. (1969). Minimum spanning trees and single linkage cluster analysis. Applied Statistics, 18(1), 54-64.

See Also

`stats::prcomp()` for principal component analysis.

Examples

```
set.seed(123)
X <- matrix(rnorm(60), nrow = 20, ncol = 3)
graph <- create.cmst.graph(X, verbose = FALSE)
graph$n_vertices
graph$n_edges
```

---

create.complete.graph *Create a Complete Graph*

Description

Create a Complete Graph

Usage

```
create.complete.graph(n)
```

Arguments

n	Integer number of vertices.

Value

An adjacency list for the complete graph on n vertices.

---

create.distance.plot *Create Distance Plot*

Description

Creates a plot of edit distances with optional smoothing or regression results.

Usage

```
create.distance.plot(
  x,
  y,
  smooth = TRUE,
  smooth.method = "loess",
  mark.x = NULL,
  xlab = "Number of Nearest Neighbors (k)",
  ylab = "Edit Distance",
  main = NULL,
  width = 6,
  height = 6,
  ...
)
```

Arguments

x	Numeric vector of x-axis values (e.g., k values).
y	Numeric vector of y-axis values (e.g., edit distances).
smooth	Logical indicating whether to add a smoothing line (default: TRUE).
smooth.method	Character string specifying the smoothing method. Options include "loess" (default), "lm", "glm", "gam".
mark.x	Optional numeric value(s) to mark with vertical lines.
xlab	Character string for x-axis label (default: "Number of Nearest Neighbors (k)").
ylab	Character string for y-axis label (default: "Edit Distance").
main	Character string for plot title (default: NULL, no title).
width	Numeric value for plot width in inches (default: 6).
height	Numeric value for plot height in inches (default: 6).
...	Additional arguments passed to `plot()`.

Details

The function creates a scatter plot with optional smoothing line. If `mark.x` is provided, vertical dashed lines are added at those x-values.

Value

Returns `NULL` invisibly after drawing the plot.

Examples

```
k.vals <- c(5, 10, 15, 20)
distances <- c(2.2, 1.4, 1.1, 1.3)
create.distance.plot(k.vals, distances, smooth = FALSE)

min.k <- k.vals[which.min(distances)]
create.distance.plot(k.vals, distances, mark.x = min.k,
                     smooth = FALSE,
                     main = "Edit Distance vs k")
```

`create.empty.graph` *Create an Empty Graph*

Description

Create an Empty Graph

Usage

```
create.empty.graph(n)
```

Arguments

n	Integer number of vertices.

Value

An adjacency list for the empty graph on n vertices.

---

create.geodesic.iknn.graph

Create a graph-geodesic iKNN graph

Description

Rebuilds an iKNN graph from an existing weighted graph by using graph geodesic distances in place of Euclidean distances. For every vertex i, the graph-metric neighbor set contains the first k finite graph-geodesic nearest vertices under the same convention used by create.iknn.graphs(). Because self-distance is zero, this includes i whenever i is finite from itself. Vertices i and j are connected in the returned graph when those two neighbor sets intersect.

Usage

```
create.geodesic.iknn.graph(graph, k)
```

Arguments

graph	A weighted graph list with entries adj_list and weight_list, as returned by create.iknn.graphs(..., compute.full = TRUE) in geom_pruned_graphs.
k	Integer number of graph-metric nearest vertices in each neighbor set, matching the k convention used by create.iknn.graphs().

Value

A list with entries:

adj_list 1-based adjacency list for the final graph.

weight_list Final graph-geodesic edge lengths from the input graph.

raw_adj_list, raw_weight_list The native geodesic iKNN graph.

pruned_adj_list, pruned_weight_list Identical to raw_* because geodesic iKNN graphs do not currently have a pruning stage.

raw_repaired_adj_list, raw_repaired_weight_list, pruned_repaired_adj_list, pruned_repaired_weight_list, repaired_ Lifecycle aliases identical to the final graph because this constructor has no MST repair or pruning stage.

isize_list Intersection sizes for each edge.

n_edges Number of undirected edges.

Examples

```
graph <- list(
  adj_list = list(2L, c(1L, 3L), 2L),
  weight_list = list(1, c(1, 1), 1)
)
create.geodesic.iknn.graph(graph, k = 1)
```

`create.grid.graph` *Create a Refined Graph with Approximately Uniform Edge Spacing*

Description

Create a Refined Graph with Approximately Uniform Edge Spacing

Usage

```
create.grid.graph(
  adj.list,
  weight.list,
  grid.size,
  start.vertex = 1L,
  snap.tolerance = 0.1
)
```

Arguments

`adj.list`	Input graph adjacency list.
`weight.list`	Edge-length list matching `adj.list`.
`grid.size`	Target grid size parameter.
`start.vertex`	Starting vertex used for breadth-first grid placement.
`snap.tolerance`	Grid snapping tolerance.

Value

A list with `adj_list`, `weight_list`, and `grid_vertices`.

`create.iknn.graphs` *Create intersection k-nearest neighbor graphs with dual pruning*

Description

For each $k \in [k_{\min}, k_{\max}]$, builds an intersection-weighted k-NN graph and applies two pruning schemes: (1) geometric (path-to-edge ratio) and (2) intersection-size. Optionally performs PCA before graph construction.

Usage

```
create.iknn.graphs(
  X,
  kmin,
  kmax,
  max.path.edge.ratio.deviation.thld = 0.1,
  path.edge.ratio.percentile = 0.5,
  threshold.percentile = 0,
  compute.full = TRUE,
  with.isize.pruning = FALSE,
  with.edge.pruning.stats = FALSE,
  pca.dim = 100,
  variance.explained = 0.99,
  knn.metric = c("euclidean", "linf.simplex"),
  linf.tol = sqrt(.Machine$double.eps),
  n.cores = 1L,
  parallel.mode = c("auto", "k", "bucket", "hybrid", "bucket.prune"),
  hybrid.batch.size = 2L,
  verbose = TRUE,
  knn.cache.path = NULL,
  knn.cache.mode = c("none", "read", "write", "readwrite")
)
```

Arguments

`X`	A numeric matrix (or object coercible to a numeric matrix) with rows = observations and columns = features.
`kmin`	Integer ≥ 1, the minimum k.
`kmax`	Integer $> k_{\min}$, the maximum k.
`max.path.edge.ratio.deviation.thld`	Numeric in $[0, 0.2)$. Geometric pruning removes an edge (i, j) when there exists an alternative path between i and j whose path/edge length ratio minus 1.0 is *less than* this threshold. This is a deviation threshold δ in $[0, 0.2)$. Internally we compare the path-to-edge ratio R to $1 + \delta$.
`path.edge.ratio.percentile`	Numeric in $[0, 1]$. Only edges with length above this percentile are considered for geometric pruning.

`threshold.percentile`
: Numeric in $[0, 0.5]$. Percentile threshold for quantile-based edge length pruning. Default is 0 (no quantile pruning). When greater than 0, edges in the top (1 - threshold.percentile) quantile by length are removed if their removal preserves graph connectivity. For example, threshold.percentile = 0.9 removes edges in the top 10\ This pruning is applied after geometric pruning and targets unusually long edges based on absolute edge lengths rather than path-to-edge ratios.

`compute.full`
: Logical controlling graph-list outputs. If `TRUE`, return `geom_pruned_graphs` (and `isize_pruned_graphs` when `with.isize.pruning = TRUE`). If `FALSE`, both graph lists are `NULL` and only summary outputs are returned (`k_statistics`, plus `edge_pruning_stats` when requested).

`with.isize.pruning`
: Logical. If `TRUE`, compute and return intersection-size pruned graphs/statistics. Default is `FALSE`.

`with.edge.pruning.stats`
: Logical. If `TRUE`, compute and return edge-level pruning statistics. Default is `FALSE`.

`pca.dim`
: Positive integer or `NULL`. If not `NULL` and `ncol(X) > pca.dim`, PCA is used to reduce to at most `pca.dim` components.

`variance.explained`
: Numeric in $(0, 1]$ or `NULL`. If not `NULL`, choose the smallest number of PCs whose cumulative variance explained exceeds this threshold, capped by `pca.dim`.

`knn.metric`
: Character scalar specifying the geometry used for kNN search. `"euclidean"` uses ordinary ambient Euclidean distance. `"linf.simplex"` uses the unfolded intrinsic metric on the L^∞-simplex and requires rows of `X` to be L^∞-normalized.

`linf.tol`
: Positive numeric tolerance used only when `knn.metric = "linf.simplex"` to identify active simplex faces and validate that `max(row) = 1`.

`n.cores`
: Integer or `NULL`. Number of CPU cores. `NULL` uses the maximum available (OpenMP build only).

`parallel.mode`
: Character execution mode for graph construction: `"auto"`, `"k"`, `"bucket"`, `"hybrid"` (or alias `"bucket.prune"`). `"hybrid"` performs bucket-parallel graph build and then k-parallel pruning in batches.

`hybrid.batch.size`
: Positive integer batch size used when `parallel.mode = "hybrid"`.

`verbose`
: Logical; print progress and timing.

`knn.cache.path`
: Optional character scalar path to a binary kNN cache file. Used only when `knn.cache.mode != "none"`.

`knn.cache.mode`
: Character cache mode:

  - `"none"`: always compute kNN; do not read/write cache.
  - `"read"`: read kNN from cache only; if cache is missing/invalid, error.
  - `"write"`: always compute kNN and atomically write/overwrite cache.
  - `"readwrite"`: read valid cache when available; if cache file is missing, compute and write cache; if cache exists but is invalid, error.

Details

Geometric pruning uses the deviation threshold `max.path.edge.ratio.deviation.thld` and the filtering percentile `path.edge.ratio.percentile`. Intersection-size pruning currently uses a maximum alternative path length of 2.

Output behavior by flags:

- `compute.full` = `FALSE`: both `geom_pruned_graphs` and `isize_pruned_graphs` are `NULL`.
- `compute.full` = `TRUE` and `with.isize.pruning` = `FALSE`: `geom_pruned_graphs` is returned and `isize_pruned_graphs` is `NULL`.

Value

A list of class `"iknn_graphs"` with entries:

k_statistics Matrix of per-k edge counts and reductions. (If the C++ side supplies column names, they're preserved. Otherwise we add names consistent with what the C++ returns.)

geom_pruned_graphs If `compute.full=TRUE`, list of geometrically pruned graphs (adjacency + weights); otherwise `NULL`.

isize_pruned_graphs If `compute.full=TRUE` and `with.isize.pruning=TRUE`, list of intersection-size pruned graphs; otherwise `NULL`.

edge_pruning_stats If `with.edge.pruning.stats=TRUE`, list (per k) of edge-level statistics; otherwise `NULL`.

Examples

```
# Generate sample data
X <- matrix(rnorm(100 * 5), 100, 5)

# Basic usage
res1 <- create.iknn.graphs(
  X, kmin = 3, kmax = 10, n.cores = 1,
  compute.full = FALSE
)

# With custom pruning parameters
res2 <- create.iknn.graphs(
  X, kmin = 3, kmax = 10,
  max.path.edge.ratio.deviation.thld = 0.1,
  path.edge.ratio.percentile = 0.5,
  compute.full = TRUE,
  n.cores = 1,
  verbose = TRUE
)

# View statistics for each k
print(res2$k_statistics)
```

`create.iterated.iknn.graphs` *Create iterated graph-geodesic iKNN graphs*

Description

Constructs `G0` with `create.iknn.graphs()`, then constructs `G1`, `G2`, ..., `Gm` by repeatedly applying `create.geodesic.iknn.graph()` to each graph in the `kmin:kmax` sequence. This implements the iterated nerve rule

$$\{i, j\} \in E(G_{t+1}) \iff kNN_{G_t}(i) \cap kNN_{G_t}(j) \neq \emptyset,$$

with edge length

$$\ell_{t+1}(i, j) = d_{G_t}(i, j).$$

Usage

```
create.iterated.iknn.graphs(
  X,
  kmin,
  kmax,
  n.iterations = 3L,
  max.path.edge.ratio.deviation.thld = 0,
  path.edge.ratio.percentile = 0.5,
  threshold.percentile = 0,
  pca.dim = 100,
  variance.explained = 0.99,
  n.cores = 1L,
  parallel.mode = c("auto", "k", "bucket", "hybrid", "bucket.prune"),
  hybrid.batch.size = 2L,
  verbose = TRUE,
  knn.cache.path = NULL,
  knn.cache.mode = c("none", "read", "write", "readwrite")
)
```

Arguments

`X`	Numeric matrix with rows as observations and columns as features.
`kmin`	Integer minimum `k`.
`kmax`	Integer maximum `k`.
`n.iterations`	Non-negative integer number of geodesic rebuilds after `G0`. The default `3` returns `G0`, `G1`, `G2`, and `G3`.

`max.path.edge.ratio.deviation.thld`, `path.edge.ratio.percentile`, `threshold.percentile`
: Initial `G0` pruning arguments forwarded to `create.iknn.graphs()`. The default deviation and quantile thresholds are zero so that `G0` is the unpruned iKNN 1-skeleton.

`pca.dim`, `variance.explained`, `n.cores`, `parallel.mode`, `hybrid.batch.size`, `verbose`, `knn.cache.path`, `knn.cache.mode`
: Additional arguments forwarded to `create.iknn.graphs()` for the initial `G0` construction.

Value

A list of class `"iterated_iknn_graphs"` with entries:

k_values Integer vector of requested `k` values.

n_iterations Number of geodesic rebuilds after `G0`.

initial_graphs The `"iknn_graphs"` object returned by `create.iknn.graphs()` for `G0`.

graphs Nested list named `G0`, `G1`, ...; each entry is a named list of graph objects for `kmin:kmax`.

summary Data frame with one row per iteration and `k`.

Examples

```
set.seed(1)
X <- matrix(rnorm(30), ncol = 2)
out <- create.iterated.iknn.graphs(X, kmin = 2, kmax = 3, n.iterations = 1,
                                   verbose = FALSE)
out$summary
```

---

create.maximal.packing

Create a Maximal Packing of Vertices in a Graph

Description

Creates a maximal packing of vertices based on a specified grid size. The algorithm places vertices in the graph using a separation distance that produces a vertex set packing of size approximately equal to the given grid size. A vertex packing is a set of vertices where each pair is separated by at least a specified distance.

Usage

```
create.maximal.packing(
  adj.list,
  weight.list,
  grid.size,
  max.iterations = 20,
  precision = 0.1
)
```

Arguments

adj.list	A list where each element is a vector of adjacent vertex indices (1-based) for the corresponding vertex. Must represent an undirected graph.
weight.list	A list where each element is a vector of edge weights corresponding to the adjacencies in adj.list. All weights must be positive.
grid.size	A positive integer (>= 2) specifying the approximate separation distance between vertices in the packing.
max.iterations	Maximum number of iterations for the algorithm to converge. Default is 20.
precision	Precision threshold for convergence of the algorithm. Must be between 0 and 0.5. Default is 0.1.

Details

The function computes a maximal packing by iteratively selecting vertices that are approximately grid.size apart from each other. The process starts from one of the graph's diameter endpoints (determined automatically) and continues until no more vertices can be added to the packing without violating the distance constraint.

```
The returned \code{graph_diameter} represents the longest shortest path
in the graph, which provides insight into the graph's overall structure
and extent.

The \code{max_packing_radius} value indicates the minimum distance that
separates any two vertices in the packing. This value is determined
through binary search to achieve a packing of approximately the
requested size.

The function validates input parameters and ensures that the graph
structure is properly specified before computing the packing. The graph
must be undirected (symmetric adjacency) and connected.
```

Value

A list with class "maximal_packing" containing five components:

adj_list	A list of adjacency vectors for each vertex in the resulting grid graph
weight_list	A list of weight vectors corresponding to each adjacency
grid_vertices	An integer vector of vertex indices that form the maximal packing
graph_diameter	A numeric value representing the maximum shortest path distance between any two vertices in the graph
max_packing_radius	A numeric value representing the optimal radius used for the final packing, which is the minimum guaranteed distance between any two vertices in the packing

See Also

validate.maximal.packing, verify.maximal.packing

Examples

```
adj.list <- list(c(2), c(1, 3), c(2, 4), c(3, 5), c(4))
weight.list <- list(c(1), c(1, 1), c(1, 1), c(1, 1), c(1))

result <- create.maximal.packing(adj.list, weight.list, grid.size = 2)
result$grid_vertices
result$max_packing_radius
```

`create.mknn.graph` *Compute a Mutual k-Nearest Neighbor Graph with Weights*

Description

Creates an undirected graph where two points are connected by an edge if and only if they are present in each other's k-nearest neighbor (kNN) lists. The weight of each edge represents the distance between the connected vertices.

Usage

```
create.mknn.graph(
  X,
  k,
  prune.method = c("none", "local.geodesic", "global.geodesic.ratio"),
  max.path.edge.ratio.deviation.thld = 0.1,
  path.edge.ratio.percentile = 0.5,
  prune.tau = 1.05,
  prune.local.k = NULL,
  with.pruned.edge.stats = FALSE,
  connect.components = FALSE,
  connect.method = c("component.mst", "component.mst.ann", "global.mst"),
  bridge.k = NULL,
  bridge.k.max = NULL,
  bridge.growth = 2
)
```

Arguments

`X`	A numeric matrix or data frame where each row represents a data point and each column represents a feature/dimension.
`k`	A positive integer specifying the number of nearest neighbors to consider. Must be at least 2.
`prune.method`	Character scalar. `"none"` disables geometric pruning. `"local.geodesic"` applies the experimental local geometric pruning stage before optional MST connectivity repair. `"global.geodesic.ratio"` applies whole-graph geodesic-ratio pruning before optional MST connectivity repair.

`max.path.edge.ratio.deviation.thld`
: Numeric scalar in `[0, 0.2)`. For `prune.method = "global.geodesic.ratio"`, an edge may be removed when the shortest alternative path is at most `1 + max.path.edge.ratio.deviati` times the direct edge length.

`path.edge.ratio.percentile`
: Numeric scalar in `[0, 1]`. For `prune.method = "global.geodesic.ratio"`, only edges at or above this edge length percentile are considered.

`prune.tau`
: Numeric scalar greater than 1. For local geometric pruning, an edge may be removed when a retained local alternative path is at most this multiplicative factor times the direct edge length.

`prune.local.k`
: Integer scalar or `NULL`. Number of nearest neighbors used to form local neighborhoods for pruning. Defaults to `k`.

`with.pruned.edge.stats`
: Logical scalar. If `TRUE`, return a data frame with one row per locally pruned edge.

`connect.components`
: Logical scalar. If `TRUE`, add MST bridge edges so the final graph is connected whenever possible.

`connect.method`
: Character scalar. `"component.mst"` adds exact shortest inter-component bridges. `"component.mst.ann"` tries sparse ANN bridge candidates before automatic exact fallback. `"global.mst"` unions the graph with the full Euclidean MST.

`bridge.k`
: Integer scalar or `NULL`. Initial ANN bridge neighborhood size for `connect.method = "component.mst.ann"`.

`bridge.k.max`
: Integer scalar or `NULL`. Maximum ANN bridge neighborhood size before exact fallback.

`bridge.growth`
: Numeric scalar greater than 1. Multiplicative growth factor for ANN bridge neighborhoods.

Details

The mutual k-nearest neighbor graph is a symmetric graph where an edge between vertices i and j exists if and only if:

- j is among the k nearest neighbors of i, AND
- i is among the k nearest neighbors of j

This mutual relationship ensures that the resulting graph is undirected and typically sparser than a standard k-nearest neighbor graph.

The final graph is always stored in `adj_list`/`weight_list`. Lifecycle diagnostic fields follow the same convention as the other graph constructors: `raw_*` is before pruning and MST repair, and `pruned_*` is after pruning but before MST repair. The `raw_repaired_*`, `pruned_repaired_*`, and `repaired_pruned_*` branches allow one constructor call to compare native, prune-first, and repair-first graph geodesic geometries.

Value

A list with class "mknn_graph" containing:

adj_list A list where element i contains the indices of vertices connected to vertex i by an edge in the final graph after optional MST component repair.

weight_list A list where element i contains the distances between vertex i and its connected neighbors in the final graph (in the same order as `adj_list[[i]]`).

raw_adj_list, raw_weight_list The native mutual-kNN graph before pruning and before optional MST component repair.

pruned_adj_list, pruned_weight_list The graph after optional local geometric pruning and before optional MST component repair. If pruning is disabled, these fields are identical to the raw graph.

raw_repaired_adj_list, raw_repaired_weight_list The native mutual kNN graph after MST component repair.

pruned_repaired_adj_list, pruned_repaired_weight_list The prune-first branch after MST component repair.

repaired_pruned_adj_list, repaired_pruned_weight_list The repair-first branch after applying local geometric pruning to the repaired raw graph.

n_vertices The number of vertices in the graph.

n_edges The total number of edges in the graph.

k The k value used to construct the graph.

See Also

`create.mknn.graphs` for creating multiple graphs with different k values, `summary.mknn_graphs` for summarizing graph properties

Examples

```
set.seed(123)
X <- matrix(rnorm(60), ncol = 2)
graph <- create.mknn.graph(X, k = 4)
graph$n_vertices
graph$n_edges
graph$adj_list[[1]]
```

---

create.mknn.graphs *Create Multiple Mutual kNN Graphs with Geometric Pruning*

Description

Constructs a series of mutual k-nearest neighbor (MkNN) graphs across a range of k values, with optional geometric pruning to remove redundant edges based on alternative path analysis.

Usage

```
create.mknn.graphs(
  X,
  kmin,
  kmax,
  max.path.edge.ratio.thld = 1.2,
  path.edge.ratio.percentile = 0.5,
  compute.full = FALSE,
  pca.dim = 100,
  variance.explained = 0.99,
  verbose = FALSE
)
```

Arguments

`X`	A numeric matrix where rows represent data points and columns represent features. Data frames will be converted to matrices.
`kmin`	Minimum k value to consider (positive integer, at least 2).
`kmax`	Maximum k value to consider (must be >= kmin).
`max.path.edge.ratio.thld`	Maximum acceptable ratio of alternative path length to direct edge length for pruning. Edges with alternative paths having ratio <= this value will be pruned. Default is 1.2. Set to 0 or negative to disable pruning.
`path.edge.ratio.percentile`	Percentile threshold (0.0-1.0) for edge lengths to consider for pruning. Only edges with length greater than this percentile are evaluated. Default is 0.5.
`compute.full`	Logical; if TRUE, returns complete graph structures for each k value. If FALSE, returns only summary statistics. Default is FALSE.
`pca.dim`	Maximum number of principal components to use if dimensionality reduction is applied. Default is 100. Set to NULL to skip dimensionality reduction.
`variance.explained`	Target percentage of variance to be explained by the principal components (0-1). Default is 0.99. If this can be achieved with fewer than pca.dim components, the smaller number is used. Set to NULL to use exactly pca.dim components.
`verbose`	Logical; if TRUE, displays progress messages. Default is FALSE.

Details

This function performs the following steps for each k value:

1. **Mutual kNN Graph Construction**: Creates a graph where edges exist only between mutually nearest neighbors.
2. **Geometric Pruning** (optional): Removes edges where alternative paths exist with acceptable length ratios. An edge (i,j) is pruned if there exists a path i->k->j such that (d(i,k) + d(k,j)) / d(i,j) <= threshold.

3. **Dimensionality Reduction** (optional): If the data has more than pca.dim dimensions, PCA is applied before graph construction to improve computational efficiency and reduce noise.

The geometric pruning step helps create sparser graphs by removing edges that can be well-approximated by two-hop paths, which can improve graph quality for clustering and visualization tasks.

Value

An object of class "mknn_graphs" containing:

k_statistics A data frame with columns: k (k value), n_edges (edges before pruning), n_edges_pruned (edges after pruning), n_removed (number of removed edges), reduction_ratio (proportion of edges removed).

pruned_graphs If compute.full=TRUE, a named list of pruned graphs for each k value. NULL otherwise.

edge_pruning_stats A list of pruning statistics for each k value, containing edge lengths and path ratios.

The returned object also has the following attributes:

- kmin, kmax: The k range used
- max.path.edge.ratio.thld: The pruning threshold used
- path.edge.ratio.percentile: The percentile threshold used
- pca: If PCA was performed, contains dimensionality reduction details

See Also

`create.mknn.graph` for creating a single MkNN graph, `summary.mknn_graphs` for summarizing the results

Examples

```
set.seed(123)
X <- matrix(rnorm(80), ncol = 2)
result <- create.mknn.graphs(
  X, kmin = 3, kmax = 4, compute.full = TRUE, verbose = FALSE
)
result$k_statistics

X.high <- matrix(rnorm(120), nrow = 20, ncol = 6)
result.pca <- create.mknn.graphs(
  X.high, kmin = 2, kmax = 3, pca.dim = 3, verbose = FALSE
)
attr(result.pca, "pca")$n_components
```

create.path.graph *Create a Path Graph with Limited Hop Distance*

Description

Create a Path Graph with Limited Hop Distance

Usage

```
create.path.graph(graph, edge.lengths, h)
```

Arguments

graph	A graph adjacency list using 1-based vertex indices.
edge.lengths	Edge-length list matching graph.
h	Integer maximum hop count.

Value

An object of class "path.graph".

create.path.graph.series *Create a Series of Path Graphs*

Description

Create a Series of Path Graphs

Usage

```
create.path.graph.series(graph, edge.lengths, h.values)
```

Arguments

graph	A graph adjacency list.
edge.lengths	Edge-length list.
h.values	Positive integer hop limits.

Value

A list of "path.graph" objects.

create.plm.graph *Create a Path Length Matrix Graph Structure*

Description

Create a Path Length Matrix Graph Structure

Usage

```
create.plm.graph(graph, edge.lengths, h)
```

Arguments

graph	Adjacency list using 1-based vertex indices.
edge.lengths	Edge-length list matching graph.
h	Odd positive integer maximum path length in hops.

Value

An object of class path.graph.plm.

create.random.graph *Create a Random Undirected Graph*

Description

Create a Random Undirected Graph

Usage

```
create.random.graph(n_vertices, avg_degree, connected = TRUE)
```

Arguments

n_vertices	Integer number of vertices.
avg_degree	Target average degree.
connected	Logical; if TRUE, first build a random spanning tree.

Value

A list with adj.list and weight.list.

`create.rknn.graph` *Compute a Radius-kNN Graph*

Description

Creates an undirected Euclidean radius graph from a numeric data matrix. Use `type = "fixed"` for a fixed-radius graph, where vertices `i` and `j` are adjacent exactly when their Euclidean distance is at most `radius`. Use `type = "adaptive.radius"` for an adaptive-radius graph based on observation-specific local scale distances.

Usage

```
create.rknn.graph(
  X,
  type = c("fixed", "adaptive.radius"),
  radius = NULL,
  k.scale = NULL,
  radius.factor = 1,
  radius.rule = c("max", "min", "geomean"),
  radius.search = c("ann", "all.pairs"),
  return.timing = FALSE,
  graph.detail = c("full", "minimal"),
  prune.method = c("none", "local.geodesic", "global.geodesic.ratio"),
  max.path.edge.ratio.deviation.thld = 0.1,
  path.edge.ratio.percentile = 0.5,
  prune.tau = 1.05,
  prune.local.k = NULL,
  with.pruned.edge.stats = FALSE,
  connect.components = FALSE,
  connect.method = c("component.mst", "component.mst.ann", "global.mst"),
  bridge.k = NULL,
  bridge.k.max = NULL,
  bridge.growth = 2
)
```

Arguments

`X`	Numeric matrix or data frame with observations in rows.
`type`	Character scalar. `"fixed"` builds a fixed-radius graph using `radius`. `"adaptive.radius"` builds an adaptive-radius graph using `k.scale`, `radius.factor`, and `radius.rule`.
`radius`	Positive numeric scalar. Edges are included when $\|x_i - x_j\|_2 \leq radius$. Required for `type = "fixed"`.
`k.scale`	Positive integer smaller than `nrow(X)`. Defines local scale distances σ_i. Required for `type = "adaptive.radius"`.
`radius.factor`	Positive numeric scalar multiplying the adaptive radius.

`radius.rule` Character scalar. `"max"` uses $d_{ij} \leq radius.factor \max(\sigma_i, \sigma_j)$; `"min"` uses $d_{ij} \leq radius.factor \min(\sigma_i, \sigma_j)$; and `"geomean"` uses $d_{ij} \leq radius.factor \sqrt{\sigma_i \sigma_j}$. The `"geomean"` rule is the continuous-kNN support rule.

`radius.search` Character scalar. `"ann"` uses the bundled ANN kd-tree to perform exact fixed-radius candidate searches before applying the pair-specific adaptive-radius rule. `"all.pairs"` uses the direct $O(n^2)$ pair scan and is retained as a reference path. Used only for `type = "adaptive.radius"`.

`return.timing` Logical scalar. If `TRUE`, attach a construction timing table for adaptive-radius graphs.

`graph.detail` Character scalar. `"full"` returns the complete graph lifecycle object for adaptive-radius graphs. `"minimal"` returns only the graph fields needed by fitting code and is currently allowed only for adaptive-radius graphs with `prune.method = "none"` and `connect.components = FALSE`.

`prune.method` Character scalar. `"none"` disables geometric pruning. `"local.geodesic"` applies the experimental local geometric pruning stage before optional MST connectivity repair. `"global.geodesic.ratio"` applies whole-graph geodesic-ratio pruning before optional MST connectivity repair.

`max.path.edge.ratio.deviation.thld`
Numeric scalar in `[0, 0.2)`. For `prune.method = "global.geodesic.ratio"`, an edge may be removed when the shortest alternative path is at most `1 + max.path.edge.ratio.deviati` times the direct edge length.

`path.edge.ratio.percentile`
Numeric scalar in `[0, 1]`. For `prune.method = "global.geodesic.ratio"`, only edges at or above this edge length percentile are considered.

`prune.tau` Numeric scalar greater than 1. For local geometric pruning, an edge may be removed when a retained local alternative path is at most this multiplicative factor times the direct edge length.

`prune.local.k` Integer scalar or `NULL`. Number of nearest neighbors used to form local neighborhoods for pruning. For fixed-radius graphs, `NULL` uses the median positive raw graph degree, clipped to `[1, n - 1]`. For adaptive-radius graphs, `NULL` defaults to `k.scale`.

`with.pruned.edge.stats`
Logical scalar. If `TRUE`, return a data frame with one row per locally pruned edge.

`connect.components`
Logical scalar. If `TRUE`, add MST bridge edges so the final graph is connected whenever possible.

`connect.method` Character scalar. `"component.mst"` adds exact shortest inter-component bridges. `"component.mst.ann"` tries sparse ANN bridge candidates before automatic exact fallback. `"global.mst"` unions the graph with the full Euclidean MST.

`bridge.k` Integer scalar or `NULL`. Initial ANN bridge neighborhood size for `connect.method = "component.mst.ann"`.

`bridge.k.max` Integer scalar or `NULL`. Maximum ANN bridge neighborhood size before exact fallback.

`bridge.growth` Numeric scalar greater than 1. Multiplicative growth factor for ANN bridge neighborhoods.

Value

For type = "fixed", a list of class "radius_graph". For type = "adaptive.radius", a list of class "adaptive_radius_graph". Both contain adjacency lists, edge weights, edge matrix, and component diagnostics. The final graph is stored in adj_list/weight_list; raw and pruned lifecycle graph stages are stored in corresponding raw_*/pruned_* fields when graph.detail = "full".

Examples

```
X <- matrix(c(0, 1, 3), ncol = 1)
create.rknn.graph(X, type = "fixed", radius = 1.1)$edge_matrix
create.rknn.graph(X, type = "adaptive.radius", k.scale = 1)$edge_matrix
```

---

create.rknn.graphs *Compute Adaptive Radius-kNN Graphs Across k Values*

Description

Creates a sequence of adaptive-radius graphs by varying the k.scale parameter passed to create.rknn.graph(type = "adaptive.radius"). By default, ANN-backed construction uses the batched C++ backend while the direct all-pairs reference path uses the scalar R loop. Set backend = "r" to force the scalar loop for ANN parity checks.

Usage

```
create.rknn.graphs(
  X,
  kmin = NULL,
  kmax = NULL,
  ...,
  k.values = NULL,
  backend = c("auto", "cpp", "r")
)
```

Arguments

X	Numeric matrix or data frame with observations in rows.
kmin, kmax	Optional integer scalars defining the inclusive k range. Required when k.values is NULL.
...	Additional arguments forwarded to create.rknn.graph() with type = "adaptive.radius". The arguments type, k.scale, and radius are reserved by this plural constructor.
k.values	Optional integer vector of k values to evaluate. When supplied, k.values is used instead of kmin:kmax, and the returned graph order follows the supplied vector.

backend — Character scalar. "auto" uses the batched C++ ANN backend when radius.search = "ann" and the scalar R loop when radius.search = "all.pairs". "cpp" forces the batched C++ ANN backend and therefore requires radius.search = "ann". "r" forces the scalar R loop and is retained as a reference parity backend.

Value

A list of class "rknn_graphs" with components:

graphs A named list of adaptive-radius graph objects, one per k.

k_statistics A data frame with one row per k containing edge, component, and MST-repair counts.

timing Present only when forwarded options request graph timing; contains backend-specific construction timing rows.

Attributes include kmin, kmax, k.values, and n_vertices.

See Also

create.rknn.graph()

Examples

```
X <- matrix(c(0, 1, 3, 4), ncol = 1)
result <- create.rknn.graphs(
  X,
  kmin = 1,
  kmax = 2,
  radius.search = "all.pairs",
  graph.detail = "minimal",
  prune.method = "none"
)
names(result$graphs)
result$k_statistics
```

---

create.single.iknn.graph

Create a Single Intersection-weighted k-Nearest Neighbors Graph

---

Description

Creates and prunes an intersection-weighted k-nearest neighbors (IWD-kNN) graph from input data. The graph is constructed based on feature space distances and intersection sizes between neighbor sets.

Usage

```
create.single.iknn.graph(
  X,
  k,
  max.path.edge.ratio.deviation.thld = 0.1,
 prune.method = c("global.geodesic", "none", "local.geodesic", "global.geodesic.ratio"),
  prune.tau = NULL,
  prune.local.k = NULL,
  path.edge.ratio.percentile = 0.5,
  threshold.percentile = 0,
  compute.full = FALSE,
  with.isize.pruning = FALSE,
  with.edge.pruning.stats = FALSE,
  pca.dim = 100,
  variance.explained = 0.99,
  knn.metric = c("euclidean", "linf.simplex"),
  linf.tol = sqrt(.Machine$double.eps),
  connect.components = FALSE,
  connect.method = c("component.mst", "component.mst.ann", "global.mst"),
  bridge.k = NULL,
  bridge.k.max = NULL,
  bridge.growth = 2,
  with.lifecycle.branches = FALSE,
  verbose = TRUE,
  knn.cache.path = NULL,
  knn.cache.mode = c("none", "read", "write", "readwrite")
)
```

Arguments

`X`	A numeric matrix where rows represent data points and columns represent features. Data frames will be coerced to matrices.
`k`	Integer scalar. The number of nearest neighbors to consider for each point. Must be positive and less than the number of data points.
`max.path.edge.ratio.deviation.thld`	Numeric in $[0, 0.2)$. Geometric pruning removes an edge (i, j) when there exists an alternative path between i and j whose path/edge length ratio minus 1.0 is *less than* this threshold. This is a deviation threshold δ in $[0, 0.2)$. Internally we compare the path-to-edge ratio R to $1 + \delta$. Used when `prune.method` is `"global.geodesic"` or `"global.geodesic.ratio"`.
`prune.method`	Character scalar. `"global.geodesic"` preserves the historical whole-graph geometric pruning behavior controlled by `max.path.edge.ratio.deviation.thld` and `path.edge.ratio.percentile`. `"global.geodesic.ratio"` is a separately named alias for that whole-graph geodesic-ratio behavior. `"local.geodesic"` applies the same experimental local geometric pruning stage used by sKNN/mKNN. `"none"` disables geometric pruning; quantile pruning controlled by `threshold.percentile` is still honored.

`prune.tau` Numeric scalar greater than 1, or `NULL`. Multiplicative threshold for local geometric pruning. If `NULL`, defaults to `max(1 + max.path.edge.ratio.deviation.thld, 1.05)`.

`prune.local.k` Integer scalar or `NULL`. Number of nearest neighbors used to form local neighborhoods for local geometric pruning. Defaults to `k`.

`path.edge.ratio.percentile`
Numeric in $[0, 1]$. Only edges with length above this percentile are considered for geometric pruning.

`threshold.percentile`
Numeric in $[0, 0.5]$. Percentile threshold for quantile-based edge length pruning. Default is 0 (no quantile pruning). When greater than 0, edges in the top (1 - threshold.percentile) quantile by length are removed if their removal preserves graph connectivity. For example, threshold.percentile = 0.9 removes edges in the top 10\ This pruning is applied after geometric pruning and targets unusually long edges based on absolute edge lengths rather than path-to-edge ratios.

`compute.full` Logical scalar controlling additional full-payload outputs. If TRUE, keeps legacy original-graph components `isize_list` and `conn_comps`. Lifecycle graph payloads are always returned: `raw_adj_list`/`raw_weight_list`, `pruned_adj_list`/`pruned_weight_list`, and final `adj_list`/`weight_list`.

`with.isize.pruning`
Logical scalar. If TRUE, compute intersection-size pruning outputs and related statistics. Default is FALSE.

`with.edge.pruning.stats`
Logical scalar. If TRUE, compute edge-level pruning statistics. Default is FALSE.

`pca.dim` Maximum number of principal components to use if dimensionality reduction is applied (default: 100). Set to NULL to skip dimensionality reduction.

`variance.explained`
Percentage of variance to be explained by the principal components (default: 0.99). If this threshold can be met with fewer components than pca.dim, the smaller number will be used. Set to NULL to use exactly pca.dim components.

`knn.metric` Character scalar specifying the geometry used for kNN search. `"euclidean"` uses ordinary ambient Euclidean distance. `"linf.simplex"` uses the unfolded intrinsic metric on the L^∞-simplex and requires rows of `X` to be L^∞-normalized.

`linf.tol` Positive numeric tolerance used only when `knn.metric = "linf.simplex"` to identify active simplex faces and validate that `max(row) = 1`.

`connect.components`
Logical scalar. If TRUE, add MST bridge edges to the final pruned graph so it is connected whenever possible. Currently supported only with `knn.metric = "euclidean"`.

`connect.method` Character scalar. `"component.mst"` adds exact shortest inter-component bridges. `"component.mst.ann"` tries sparse ANN bridge candidates before automatic exact fallback. `"global.mst"` unions the graph with the full Euclidean MST.

`bridge.k` Integer scalar or NULL. Initial ANN bridge neighborhood size for `connect.method = "component.mst.ann"`.

`bridge.k.max` Integer scalar or NULL. Maximum ANN bridge neighborhood size before exact fallback.

`bridge.growth` Numeric scalar greater than 1. Multiplicative growth factor for ANN bridge neighborhoods.

`with.lifecycle.branches`
Logical scalar. If TRUE, compute the additional repaired lifecycle branches `raw_repaired_*`, `pruned_repaired_*`, and `repaired_pruned_*`. This is useful for graph-geodesic reconstruction experiments but can be expensive for the historical global iKNN pruning path, so the default is FALSE.

`verbose` Logical. If TRUE, print progress messages. Default is TRUE.

`knn.cache.path` Optional character scalar path to a binary kNN cache file. Used only when `knn.cache.mode != "none"`.

`knn.cache.mode` Character cache mode:

- `"none"`: always compute kNN; do not read/write cache.
- `"read"`: read kNN from cache only; if cache is missing/invalid, error.
- `"write"`: always compute kNN and atomically write/overwrite cache.
- `"readwrite"`: read valid cache when available; if cache file is missing, compute and write cache; if cache exists but is invalid, error.

Details

`adj_list` and `weight_list` always contain the final graph used by downstream algorithms. `raw_*` fields contain the native graph before pruning, and `pruned_*` fields contain the graph after pruning but before optional MST component repair. If pruning is disabled, `pruned_*` is identical to `raw_*`. The repaired lifecycle branches allow direct comparison of raw, prune-first, and repair-first graph geodesic distances from a single constructor call.

Value

An object of class "IkNN" (inheriting from "list") containing:

adj_list Adjacency lists for the final graph after optional MST repair

weight_list Edge weights for the final graph

raw_adj_list Adjacency lists for the native graph before pruning

raw_weight_list Edge weights for the native graph before pruning

pruned_adj_list Adjacency lists after pruning and before MST repair

pruned_weight_list Edge weights after pruning and before MST repair

raw_repaired_adj_list, raw_repaired_weight_list The native graph after MST component repair.

pruned_repaired_adj_list, pruned_repaired_weight_list The prune-first branch after MST component repair.

repaired_pruned_adj_list, repaired_pruned_weight_list The repair-first branch after applying the selected pruning method to the repaired raw graph.

n_edges Total number of edges in original graph

n_edges_in_pruned_graph Number of edges after pruning

n_removed_edges Number of edges removed by pruning

edge_reduction_ratio Proportion of edges removed

call The matched function call

k Number of nearest neighbors used

n Number of data points

d Number of features

If compute.full = TRUE, additional components include:

isize_list Intersection sizes for original edges

conn_comps Connected components identification

connected_components Alternative format of connected components

Examples

```
# Create sample data
set.seed(123)
X <- matrix(rnorm(100), ncol = 2)
result <- create.single.iknn.graph(X, k = 3)
summary(result)
```

`create.sknn.graph` *Compute a Symmetric k-Nearest Neighbor Graph*

Description

Creates an undirected symmetric k-nearest neighbor (sKNN) graph from a numeric data matrix. An edge between vertices `i` and `j` is present when either `j` is among the `k` nearest neighbors of `i`, or `i` is among the `k` nearest neighbors of `j`.

Usage

```
create.sknn.graph(
  X,
  k,
  prune.edges = FALSE,
  prune.method = c("none", "local.geodesic", "global.geodesic.ratio"),
  max.path.edge.ratio.deviation.thld = 0.1,
  path.edge.ratio.percentile = 0.5,
  prune.tau = 1.05,
  prune.local.k = NULL,
  with.pruned.edge.stats = FALSE,
  connect.components = FALSE,
  connect.method = c("component.mst", "component.mst.ann", "global.mst"),
  edge.weight = c("distance"),
```

```
  neighbor.method = c("exact", "ann"),
  ann.eps = 0,
  bridge.k = NULL,
  bridge.k.max = NULL,
  bridge.growth = 2
)
```

Arguments

X	Numeric matrix or data frame with observations in rows.
k	Integer scalar. Number of non-self nearest neighbors.
prune.edges	Logical scalar. If TRUE, apply experimental local geometric edge pruning before optional MST connectivity repair.
prune.method	Character scalar. "none" disables pruning. "local.geodesic" applies experimental local geometric edge pruning. "global.geodesic.ratio" applies whole-graph geodesic-ratio pruning.
max.path.edge.ratio.deviation.thld	Numeric scalar in [0, 0.2). For prune.method = "global.geodesic.ratio", an edge may be removed when the shortest alternative path is at most 1 + max.path.edge.ratio.deviati times the direct edge length.
path.edge.ratio.percentile	Numeric scalar in [0, 1]. For prune.method = "global.geodesic.ratio", only edges at or above this edge length percentile are considered.
prune.tau	Numeric scalar greater than 1. An edge may be pruned when the shortest retained local alternative path is at most this multiplicative factor times the direct edge length.
prune.local.k	Integer scalar or NULL. Number of nearest neighbors used to form local neighborhoods for pruning. Defaults to k.
with.pruned.edge.stats	Logical scalar. If TRUE, return a data frame with one row per pruned edge and columns for direct length, replacement path length, and their ratio.
connect.components	Logical scalar. If TRUE, add MST bridge edges so that the final graph is connected whenever possible.
connect.method	Character scalar. "component.mst" adds the minimum number of shortest inter-component bridges. "component.mst.ann" tries sparse ANN bridge candidates before automatic exact fallback. "global.mst" unions the sKNN graph with the Euclidean MST on all vertices.
edge.weight	Character scalar. Currently only "distance" is supported.
neighbor.method	Character scalar. "exact" uses all pairwise distances for kNN construction. "ann" uses the bundled ANN kd-tree backend for the kNN construction itself.
ann.eps	Non-negative numeric scalar reserved for approximate ANN search. Currently only 0 is supported.

bridge.k	Integer scalar or NULL. Initial number of ANN neighbors to use for sparse inter-component bridge search when connect.method = "component.mst.ann". Defaults to max(k, 10), capped at nrow(X) - 1.
bridge.k.max	Integer scalar or NULL. Maximum number of ANN neighbors to use before exact fallback. Defaults to min(nrow(X) - 1, max(50, 8 * k, bridge.k)).
bridge.growth	Numeric scalar greater than 1. Multiplicative growth factor for bridge candidate neighborhoods.

Details

With no nearest-neighbor ties, the sKNN edge rule is equivalent to

$$d_{ij} \leq \max(\sigma_i, \sigma_j),$$

where σ_i is the distance from point i to its k-th non-self nearest neighbor.

If connect.components = TRUE, the graph is augmented with MST edges. The default connect.method = "component.mst" first collapses each connected component of the sKNN graph to one component vertex, weights each component pair by the shortest data-space edge between them, and adds the minimum spanning tree of this component graph. This adds exactly m - 1 bridge edges when the original sKNN graph has m connected components. connect.method = "component.mst.ann" uses ANN nearest-neighbor bridge candidates first and automatically falls back to exact component MST when the sparse candidate graph does not connect all components. The alternative connect.method = "global.mst" unions the sKNN graph with the full Euclidean MST on the data points.

neighbor.method = "exact" computes all pairwise distances and is the deterministic reference implementation. neighbor.method = "ann" uses the bundled ANN kd-tree for the Euclidean kNN search step; exact parity can differ only in nearest-neighbor tie cases. Edge weights are always returned as ordinary Euclidean distances. Exact component.mst repair uses exact pairwise bridge distances; component.mst.ann uses ANN bridge candidates before guaranteed exact fallback.

If prune.method = "local.geodesic" or prune.edges = TRUE, an experimental local-geodesic pruning stage is applied after the sKNN support is built and before optional MST connectivity repair. Candidate edges are processed longest-first. For an edge (i, j), the edge is removed only when the currently retained graph contains an alternative path from i to j, restricted to the union of their local neighborhoods, whose length is at most prune.tau times the direct edge length. prune.method = "global.geodesic.ratio" instead uses the whole graph as the alternative-path search domain and removes long edges whose shortest alternative path is within the configured geodesic-ratio threshold.

Value

A list of class "sknn_graph" with adjacency lists, weights, edge matrix, kNN index matrix, component diagnostics, and any MST edges added. The graph lifecycle fields are:

raw_adj_list, raw_weight_list The native sKNN graph before pruning and before optional MST component repair.

pruned_adj_list, pruned_weight_list The graph after optional local geometric pruning and before optional MST component repair. If pruning is disabled, these fields are identical to the raw graph.

raw_repaired_adj_list, raw_repaired_weight_list The native sKNN graph after MST component repair.

pruned_repaired_adj_list, pruned_repaired_weight_list The prune-first branch after MST component repair.

repaired_pruned_adj_list, repaired_pruned_weight_list The repair-first branch after applying local geometric pruning to the repaired raw graph.

adj_list, weight_list The final graph after optional MST component repair. Downstream algorithms should use these fields unless they explicitly need a lifecycle diagnostic stage.

Examples

```
X <- rbind(c(0, 0), c(1, 0), c(10, 0), c(11, 0))
g <- create.sknn.graph(X, k = 1, connect.components = TRUE)
g$n_components_before
g$n_components_after
g$mst_edge_matrix
```

`create.star.graph` *Create a Star Graph by Joining Chains*

Description

Create a Star Graph by Joining Chains

Usage

```
create.star.graph(sizes)
```

Arguments

`sizes`	Positive integer chain lengths attached to the central vertex.

Value

An adjacency list for the star graph.

`create.subgraph` *Create a Subgraph from a Graph Object*

Description

Create a Subgraph from a Graph Object

Usage

```
create.subgraph(
  S.graph,
  id.indices = NULL,
  ids = NULL,
  S = NULL,
  use.sequential.indices = FALSE
)
```

Arguments

`S.graph`	List containing `adj_list` and `dist_list`.
`id.indices`	Optional vertex indices to keep.
`ids`	Optional vertex IDs to keep.
`S`	Data frame or matrix whose row names map `ids` to vertex indices.
`use.sequential.indices`	Logical; if `TRUE`, renumber kept vertices from 1 to `length(id.indices)`.

Value

A list with subgraph `adj_list` and `dist_list`.

`create.threshold.distance.graph`
Create a Threshold Distance Graph

Description

Create a Threshold Distance Graph

Usage

```
create.threshold.distance.graph(dist.matrix, threshold, include.names = TRUE)
```

Arguments

dist.matrix	Symmetric distance matrix.
threshold	Numeric distance threshold. Vertices with distance less than this value are connected.
include.names	Logical; if TRUE, preserve row names on adjacency and weight lists.

Value

A list with adj_list and weight_list.

---

deprecated-radius-graph-constructors
Deprecated Radius Graph Constructors

Description

create.radius.graph() and create.adaptive.radius.graph() are deprecated compatibility wrappers. Use create.rknn.graph() with type = "fixed" or type = "adaptive.radius" instead.

Usage

```
create.radius.graph(
  X,
  radius,
  prune.method = c("none", "local.geodesic", "global.geodesic.ratio"),
  max.path.edge.ratio.deviation.thld = 0.1,
  path.edge.ratio.percentile = 0.5,
  prune.tau = 1.05,
  prune.local.k = NULL,
  with.pruned.edge.stats = FALSE,
  connect.components = FALSE,
  connect.method = c("component.mst", "component.mst.ann", "global.mst"),
  bridge.k = NULL,
  bridge.k.max = NULL,
  bridge.growth = 2
)

create.adaptive.radius.graph(
  X,
  k.scale,
  radius.factor = 1,
  radius.rule = c("max", "min", "geomean"),
  radius.search = c("ann", "all.pairs"),
  return.timing = FALSE,
  graph.detail = c("full", "minimal"),
```

```
  prune.method = c("none", "local.geodesic", "global.geodesic.ratio"),
  max.path.edge.ratio.deviation.thld = 0.1,
  path.edge.ratio.percentile = 0.5,
  prune.tau = 1.05,
  prune.local.k = NULL,
  with.pruned.edge.stats = FALSE,
  connect.components = FALSE,
  connect.method = c("component.mst", "component.mst.ann", "global.mst"),
  bridge.k = NULL,
  bridge.k.max = NULL,
  bridge.growth = 2
)
```

Arguments

`X`	Numeric matrix or data frame with observations in rows.
`radius`	Fixed radius passed to `create.rknn.graph(type = "fixed")`.
`prune.method`, `max.path.edge.ratio.deviation.thld`, `path.edge.ratio.percentile`, `prune.tau`, `prune.local.k`, `with.pruned.edge.stats`	Geometric pruning controls passed to `create.rknn.graph()`.
`connect.components`, `connect.method`, `bridge.k`, `bridge.k.max`, `bridge.growth`	Component repair controls passed to `create.rknn.graph()`.
`k.scale`, `radius.factor`, `radius.rule`, `radius.search`	Adaptive-radius parameters passed to `create.rknn.graph(type = "adaptive.radius")`.
`return.timing`, `graph.detail`	Adaptive-radius output controls passed to `create.rknn.graph()`.

Value

A graph list produced by `create.rknn.graph()`. The fixed-radius wrapper returns an object of class `"radius_graph"`; the adaptive-radius wrapper returns an object of class `"adaptive_radius_graph"`. Each object contains the final adjacency and weight lists, an edge matrix and edge weights, graph-construction parameters, and component, pruning, and repair diagnostics.

Examples

```
X <- matrix(c(0, 1, 3), ncol = 1)
create.rknn.graph(X, type = "fixed", radius = 1.1)$edge_matrix
create.rknn.graph(X, type = "adaptive.radius", k.scale = 1)$edge_matrix
```

`detect.graph.endpoints` *Detect Graph Endpoints from a 3D Embedding*

Description

Detects terminal arm tips in a graph by:

1. computing graph endpoint scores from the 3D embedding,
2. optionally smoothing those scores on the graph using `fit.rdgraph.regression()` and `refit.rdgraph.regression(`
3. finding graph-local maxima with `detect.local.extrema()`,
4. retaining only maxima that persist across neighborhood scales.

The function works with terminal regions in an embedding; returned endpoints need not be degree-1 vertices.

Usage

```
detect.graph.endpoints(
  adj.list,
  weight.list,
  layout.3d,
  neighborhood = c("geodesic_k", "geodesic_radius"),
  k = c(10L, 20L, 30L),
  radius = NULL,
  q = 0.1,
  neighbor.weighting = c("uniform", "inverse_distance", "gaussian"),
  gaussian.sigma = NULL,
  min.neighborhood.size = 3L,
  scale.aggregate = c("mean", "median", "min", "max"),
  score.metric = c("score", "m", "s.q", "s.min"),
  smooth = FALSE,
  fitted.model = NULL,
  smooth.fit.args = NULL,
  smooth.refit.args = NULL,
  detect.max.radius = 2,
  detect.min.neighborhood.size = 2L,
  scale.stability.radius = NULL,
  min.scale.stability = 0.5,
  min.score.quantile = 0.8,
  max.endpoints = NULL,
  verbose = FALSE
)
```

Arguments

adj.list	Graph adjacency list (1-based vertex indices).
weight.list	Edge-length list aligned with adj.list.
layout.3d	Numeric matrix or data frame with 3 columns giving the 3D embedding coordinates for graph vertices.
neighborhood	Character string. Either "geodesic_k" (default) or "geodesic_radius".
k	Integer vector of geodesic neighborhood sizes used when neighborhood = "geodesic_k".
radius	Positive numeric vector of geodesic radii used when neighborhood = "geodesic_radius".
q	Quantile used for s.q. Must lie in [0, 1]. Default is 0.1.
neighbor.weighting	Character string controlling neighbor weights for m and weighted pair quantiles. One of "uniform" (default), "inverse_distance", or "gaussian".
gaussian.sigma	Optional positive numeric scale used when neighbor.weighting = "gaussian".
min.neighborhood.size	Minimum number of usable neighbors required after removing zero-length embedding vectors.
scale.aggregate	How to aggregate scores across multiple neighborhood scales. One of "mean" (default), "median", "min", or "max".
score.metric	Which score to maximize. One of "score" (default), "m", "s.q", or "s.min".
smooth	Logical; if TRUE, smooths aggregated scores on the graph before detecting extrema.
fitted.model	Optional fitted graph smoother from fit.rdgraph.regression(). If supplied, smoothing reuses its spectral decomposition via refit.rdgraph.regression().
smooth.fit.args	Optional named list of arguments passed to fit.rdgraph.regression() when smooth = TRUE and fitted.model is NULL. X, y, adj.list, weight.list, and verbose.level are filled automatically when not supplied.
smooth.refit.args	Optional named list of arguments passed to refit.rdgraph.regression() when smoothing aggregated score fields.
detect.max.radius	Maximum radius passed to detect.local.extrema() when calling endpoint candidates from the final detection score.
detect.min.neighborhood.size	Minimum neighborhood size passed to detect.local.extrema() when calling endpoint candidates.
scale.stability.radius	Optional non-negative geodesic radius used when translating per-scale local maxima into multiscale support. If NULL (default), uses detect.max.radius. This allows smoothed maxima that shift slightly inward to remain supported by nearby raw multiscale maxima.

`min.scale.stability`
: Minimum fraction of scales for which a vertex must be a local maximum of the per-scale detection score to be retained as an endpoint. Default is `0.5`.

`min.score.quantile`
: Quantile filter applied to the final detection score before returning endpoints. Default is `0.8`, which suppresses weak local maxima while keeping high-scoring terminal tips.

`max.endpoints`
: Optional positive integer cap on returned endpoints.

`verbose`
: Logical; if `TRUE`, prints progress.

Value

A list with endpoint vertices, score diagnostics, optional smoothed scores, scale stability summaries, and the local-extrema fit used for endpoint calling.

`detect.local.extrema` *Detect Local Extrema in a Graph*

Description

Detect Local Extrema in a Graph

Usage

```
detect.local.extrema(
  adj.list,
  weight.list,
  y,
  max.radius,
  min.neighborhood.size,
  detect.maxima = TRUE,
  custom.prefix = NULL
)
```

Arguments

`adj.list`
: Graph adjacency list using 1-based vertex indices.

`weight.list`
: Edge-length list aligned with `adj.list`.

`y`
: Numeric vertex function values.

`max.radius`
: Maximum graph-geodesic radius for neighborhoods.

`min.neighborhood.size`
: Minimum neighborhood size required.

`detect.maxima`
: Logical; if `TRUE`, detect maxima, otherwise minima.

`custom.prefix`
: Optional label prefix.

Value

A `local_extrema` object.

`dist.to.knn` *Produce k-NN Distance and Index Matrices from a Distance Matrix*

Description

Produce k-NN Distance and Index Matrices from a Distance Matrix

Usage

```
dist.to.knn(d, k)
```

Arguments

`d`	Symmetric distance matrix.
`k`	Number of nearest neighbors to return per row.

Value

A list with `nn.i` and `nn.d` matrices.

`edge.diff` *Compute Edge Difference Between Two Graphs*

Description

Compute Edge Difference Between Two Graphs

Usage

```
edge.diff(graph1, graph2)
```

Arguments

`graph1`	First adjacency list.
`graph2`	Second adjacency list.

Value

A list containing neighbors present in `graph1` but not `graph2` for each vertex.

estimate.geodesic.distances *Estimate Pairwise Geodesic Distances*

Description

Estimate Pairwise Geodesic Distances

Usage

```
estimate.geodesic.distances(points, k = 5, graph = NULL, method = "knn.graph")
```

Arguments

points	Numeric matrix or data frame with points in rows.
k	Positive integer k for k-NN graph construction.
graph	Optional igraph object to use directly.
method	Graph construction method, "knn.graph" or "mst".

Value

Numeric matrix of graph shortest-path distances.

euclidean.distance *Euclidean Distance Between Two Points*

Description

Euclidean Distance Between Two Points

Usage

```
euclidean.distance(p1, p2)
```

Arguments

p1	First numeric point.
p2	Second numeric point.

Value

Euclidean distance between p1 and p2.

extract.edge.lengths *Extract Unique Edge Lengths from an Undirected Graph*

Description

Extract Unique Edge Lengths from an Undirected Graph

Usage

```
extract.edge.lengths(
  adj.list,
  edge.length.list,
  method = c("vectorized", "preallocate", "parallel"),
  mc.cores = 2
)
```

Arguments

adj.list	Adjacency list.
edge.length.list	Edge-length list aligned with adj.list.
method	Extraction method: "vectorized", "preallocate", or "parallel".
mc.cores	Number of cores for method = "parallel".

Value

Numeric vector of unique undirected edge lengths.

extract.trajectory.edge.lengths *Extract Edge Lengths Along a Graph Path*

Description

Extract Edge Lengths Along a Graph Path

Usage

```
extract.trajectory.edge.lengths(
  traj,
  adj.list,
  edge.length.list,
  add.quantiles = FALSE
)
```

Arguments

`traj`	Numeric or integer vector of consecutive graph vertices.
`adj.list`	Adjacency list.
`edge.length.list`	Edge-length list aligned with `adj.list`.
`add.quantiles`	Logical; if `TRUE`, add edge-length empirical quantiles.

Value

Data frame with consecutive vertex pairs and their edge lengths.

`generate.circle.graph` *Generate a Weighted Circle Graph*

Description

Generate a Weighted Circle Graph

Usage

```
generate.circle.graph(n, type = "random", seed = NULL)
```

Arguments

`n`	Number of vertices.
`type`	Angle distribution, either `"random"` or `"uniform"`.
`seed`	Optional random seed used when `type = "random"`.

Value

A list with `adj.list` and `weight.list`.

`geodesic.core.endpoints` *Select Graph Endpoints by Core-Eccentricity Geometry*

Description

Select Graph Endpoints by Core-Eccentricity Geometry

Usage

```
geodesic.core.endpoints(
  adj.list,
  weight.list,
  core.quantile = 0.1,
  endpoint.quantile = 0.9,
  use.approx.eccentricity = TRUE,
  n.landmarks = 64L,
  max.endpoints = NULL,
  seed = 1L,
  verbose = FALSE
)
```

Arguments

`adj.list`	Graph adjacency list using 1-based vertex indices.
`weight.list`	Edge-length list aligned with `adj.list`.
`core.quantile`	Numeric in `(0, 1)` defining the low-eccentricity core.
`endpoint.quantile`	Numeric in `[0, 1]` for endpoint candidate scores.
`use.approx.eccentricity`	Use landmark-based eccentricity approximation.
`n.landmarks`	Number of landmarks when approximation is used.
`max.endpoints`	Optional positive cap on returned endpoints.
`seed`	Integer seed for landmark initialization.
`verbose`	Print backend progress.

Value

A `geodesic_core_endpoints` list of endpoints and diagnostics.

`geodesic.disk` *Geodesic Disk in a Weighted Graph*

Description

Geodesic Disk in a Weighted Graph

Usage

```
geodesic.disk(adj.list, weight.list, center.vertex, radius = NULL, n = NULL)
```

Arguments

adj.list	Adjacency list.
weight.list	Edge-length list aligned with adj.list.
center.vertex	Center vertex.
radius	Optional geodesic radius.
n	Optional target number of reachable vertices.

Value

List with vertices, effective radius, and aligned dists.

---

geodesic.knn *Estimate Geodesic Nearest Neighbors Within a Point Cloud*

Description

Estimate Geodesic Nearest Neighbors Within a Point Cloud

Usage

```
geodesic.knn(X, k, K = 5, G = NULL)
```

Arguments

X	Numeric matrix of observations.
k	Number of nearest neighbors to return.
K	Number of neighbors used to build the graph.
G	Optional legacy graph argument passed to estimate.geodesic.distances.

Value

A list with nn.index and nn.dist matrices.

geodesic.knnx *Estimate Geodesic Nearest Neighbors from Grid Points to Data Points*

Description

Estimate Geodesic Nearest Neighbors from Grid Points to Data Points

Usage

```
geodesic.knnx(X, X.grid, k, method = "knn.graph", K = 5)
```

Arguments

X	Numeric data matrix.
X.grid	Numeric grid matrix associated with X.
k	Number of nearest data neighbors returned for each grid point.
method	Legacy graph construction method argument.
K	Legacy graph neighbor argument. The implementation resets this to 2^ncol(X).

Value

A list with graph vertices, graph edges, nn.index, and nn.dist.

get.edge.weights *Get Unique Edge Weights from a Weighted Graph*

Description

Get Unique Edge Weights from a Weighted Graph

Usage

```
get.edge.weights(adj.list, weight.list, n.cores = 1L)
```

Arguments

adj.list	Adjacency list.
weight.list	Edge-weight list aligned with adj.list.
n.cores	Number of worker processes used to extract edge weights.

Value

Numeric vector of unique undirected edge weights.

get.shortest.path *Get Shortest Path Between Two Vertices*

Description

Get Shortest Path Between Two Vertices

Usage

```
get.shortest.path(pg, from, to)
```

Arguments

pg	A "path.graph" object.
from	Source vertex.
to	Target vertex.

Value

Path information, or NULL when no stored path exists.

graph.adj.mat *Convert Coordinates and Edges to a Weighted Adjacency Matrix*

Description

Convert Coordinates and Edges to a Weighted Adjacency Matrix

Usage

```
graph.adj.mat(X, E)
```

Arguments

X	Numeric coordinate matrix.
E	Two-column matrix of 1-based edge endpoints.

Value

Symmetric weighted adjacency matrix with Euclidean edge lengths.

graph.connected.components *Assign Vertices to Connected Components*

Description

Assign Vertices to Connected Components

Usage

```
graph.connected.components(adj.list)
```

Arguments

adj.list	A graph adjacency list using 1-based vertex indices.

Value

Integer vector of component IDs, one per vertex.

graph.edit.distance *Calculate Graph Edit Distance (Pure R Implementation)*

Description

Computes the graph edit distance between two weighted graphs with identical vertex sets using a pure R implementation. This function considers edge insertions, deletions, and weight modifications.

Usage

```
graph.edit.distance(
  graph1.adj.list,
  graph1.weights,
  graph2.adj.list,
  graph2.weights,
  edge.cost = 1,
  weight.cost.factor = 0.1
)
```

Arguments

`graph1.adj.list`	A list of integer vectors representing the adjacency list of the first graph. Each element contains the neighbors of the corresponding vertex.
`graph1.weights`	A list of numeric vectors representing the edge weights of the first graph. Must have the same structure as `graph1.adj.list`.
`graph2.adj.list`	A list of integer vectors representing the adjacency list of the second graph.
`graph2.weights`	A list of numeric vectors representing the edge weights of the second graph.
`edge.cost`	A positive numeric scalar representing the cost of adding or removing an edge (default: 1).
`weight.cost.factor`	A non-negative numeric scalar representing the factor to scale weight differences (default: 0.1).

Details

The graph edit distance is calculated as:

- For edges present in one graph but not the other: `edge.cost`
- For edges present in both graphs with different weights: `abs(weight1 - weight2) * weight.cost.factor`

Both graphs must have the same number of vertices. The function assumes undirected graphs.

Value

A non-negative numeric value representing the graph edit distance between the two input graphs.

Examples

```
# Create two simple graphs
graph1.adj <- list(c(2, 3), c(1, 3), c(1, 2))
graph1.wts <- list(c(1, 2), c(1, 3), c(2, 3))

graph2.adj <- list(c(2, 3), c(1, 3), c(1, 2))
graph2.wts <- list(c(1.5, 2), c(1.5, 2.5), c(2, 2.5))

# Calculate distance
dist <- graph.edit.distance(graph1.adj, graph1.wts,
                            graph2.adj, graph2.wts)
print(dist)  # Weight differences only
```

graph.embedding *Embed Graph in 2D or 3D Space*

Description

Embed Graph in 2D or 3D Space

Usage

```
graph.embedding(
  adj.list,
  weights.list = NULL,
  invert.weights = TRUE,
  dim = 2,
  method = c("fr", "kk"),
  verbose = FALSE
)
```

Arguments

adj.list	Adjacency list representation of a graph.
weights.list	Optional edge-weight list aligned with adj.list.
invert.weights	Logical; invert weights for Fruchterman-Reingold layout.
dim	Embedding dimension, either 2 or 3.
method	Layout method, "fr" or "kk".
verbose	Logical; print timing messages.

Value

Numeric layout matrix with one row per embedded vertex.

graph.geodesic.distances *Compute Graph Geodesic Distances from a Graph Object*

Description

Compute Graph Geodesic Distances from a Graph Object

Usage

```
graph.geodesic.distances(graph, vertices = NULL, stage = "final")
```

Arguments

graph	A supported graph object.
vertices	Optional 1-based vertex subset.
stage	Graph lifecycle stage.

Value

A numeric matrix of shortest-path distances.

---

graph.spectral.embedding *Generate Spectral Embedding of a Graph*

Description

Generate Spectral Embedding of a Graph

Usage

```
graph.spectral.embedding(evectors, dim, evalues = NULL)
```

Arguments

evectors	Numeric matrix of graph Laplacian eigenvectors.
dim	Embedding dimension.
evalues	Optional eigenvalues used for scaling.

Value

Numeric spectral embedding matrix.

---

graph.spectrum *Compute Graph Spectrum*

Description

Compute Graph Spectrum

Usage

```
graph.spectrum(
  graph,
  nev = NULL,
  use.R = FALSE,
  return.Laplacian = FALSE,
  return.dense = FALSE
)
```

Arguments

graph	Graph adjacency list.
nev	Number of nontrivial eigenvalues/eigenvectors to compute.
use.R	Logical; use R/igraph implementation instead of native backend.
return.Laplacian	Logical; include the graph Laplacian in the result.
return.dense	Logical; return a dense Laplacian when requested.

Value

A list with evalues, evectors, and optionally laplacian.

---

graph.summary.divergence

Compute divergence between two graphs using a selected graph summary

---

Description

Extracts one selected graph summary from two graph-like objects and computes a divergence between the resulting probability mass functions.

Currently the supported divergence is Jensen-Shannon divergence, implemented via jensen.shannon.divergence().

Usage

```
graph.summary.divergence(
  g1,
  g2,
  summary = c("degree_distribution", "edge_weight_distribution",
    "component_size_distribution", "neighborhood_label_distribution"),
  divergence = c("js"),
  labels = NULL,
  summary.args = list(),
  support = NULL,
  return.details = TRUE
)
```

Arguments

g1	First graph-like object.
g2	Second graph-like object.
summary	Summary type to compare.
divergence	Divergence type. Currently only "js".
labels	Optional shared vertex-label vector used for "neighborhood_label_distribution".

summary.args	Optional named list forwarded to compute.graph.summary.pmf().
support	Optional common support. If NULL, the union of the two PMF supports is used.
return.details	Logical; if TRUE, return a structured list.

Value

If return.details = TRUE, a list with fields:

- summary: summary type used.
- divergence: divergence type used.
- value: scalar divergence value.
- pmf1: aligned PMF for g1.
- pmf2: aligned PMF for g2.
- support: aligned support.
- metadata: summary-specific metadata for both graphs.

Otherwise, returns the scalar divergence value.

---

identical.vertex.set.weighted.graph.similarity

Weighted Graph Distance Between Graphs with Identical Vertex Sets

---

Description

Weighted Graph Distance Between Graphs with Identical Vertex Sets

Usage

```
identical.vertex.set.weighted.graph.similarity(
  graph1.adj.list,
  graph1.weights,
  graph2.adj.list,
  graph2.weights,
  calculate.normalized.deviation = FALSE
)
```

Arguments

graph1.adj.list	First graph adjacency list.
graph1.weights	First graph edge weights.
graph2.adj.list	Second graph adjacency list.
graph2.weights	Second graph edge weights.
calculate.normalized.deviation	Logical; if TRUE, normalize the L1 distance-matrix deviation.

Value

Numeric distance-matrix deviation.

`isometry.distance.correlations` *Compute Distance Preservation Correlations*

Description

Computes Pearson and Spearman correlations between estimated and true pairwise distances over the upper-triangular distance entries.

Usage

```
isometry.distance.correlations(
  D.estimated,
  D.true,
  true.tol = sqrt(.Machine$double.eps)
)
```

Arguments

`D.estimated`	Estimated distance matrix or `dist` object.
`D.true`	Ground-truth distance matrix or `dist` object.
`true.tol`	Non-negative tolerance. Pairs with true distance less than or equal to this value are excluded from relative metrics.

Value

Named numeric vector with entries `pearson_cor` and `spearman_cor`.

Examples

```
D <- as.matrix(dist(1:4))
isometry.distance.correlations(2 * D, D)
```

`isometry.distortion.quantiles` *Compute Multiplicative Distortion Quantiles*

Description

Computes quantiles of the calibrated pairwise distortion

$$\rho_{ij} = \alpha \hat{D}_{ij} / D_{ij}.$$

Values near 1 indicate approximately isometric distance preservation.

Usage

```
isometry.distortion.quantiles(
  D.estimated,
  D.true,
  probs = c(0.05, 0.5, 0.95),
  scale = TRUE,
  true.tol = sqrt(.Machine$double.eps)
)
```

Arguments

`D.estimated`	Estimated distance matrix or `dist` object.
`D.true`	Ground-truth distance matrix or `dist` object.
`probs`	Numeric vector of probabilities passed to `stats::quantile()`.
`scale`	Logical scalar. If `TRUE`, use `isometry.scale()` to optimally rescale estimated distances before computing the error.
`true.tol`	Non-negative tolerance. Pairs with true distance less than or equal to this value are excluded from relative metrics.

Value

Named numeric vector of distortion quantiles.

Examples

```
D <- as.matrix(dist(1:4))
isometry.distortion.quantiles(1.1 * D, D)
```

`isometry.geodesic.diagnostics` *Compute Geodesic-Isometry Diagnostics*

Description

Computes signed and scale-regime diagnostics for comparing estimated graph geodesic distances with reference geodesic distances. Distances are first optionally calibrated by the least-squares scale from `isometry.scale()`.

Usage

```
isometry.geodesic.diagnostics(
  D.estimated,
  D.true,
  scale = TRUE,
  true.tol = sqrt(.Machine$double.eps),
  band.probs = c(1/3, 2/3)
)
```

Arguments

`D.estimated`	Estimated distance matrix or `dist` object.
`D.true`	Ground-truth distance matrix or `dist` object.
`scale`	Logical scalar. If `TRUE`, use `isometry.scale()` to optimally rescale estimated distances before computing the error.
`true.tol`	Non-negative tolerance. Pairs with true distance less than or equal to this value are excluded from relative metrics.
`band.probs`	Numeric vector of two probabilities used to split reference distances into short, middle, and long distance bands.

Value

Named numeric vector with relative geodesic stress, signed residual bias, shortcut fraction, relative absolute residual quantiles, and median signed relative residuals in short/mid/long reference-distance bands.

Examples

```
D <- as.matrix(dist(1:4))
isometry.geodesic.diagnostics(1.1 * D, D)
```

`isometry.rel.abs.error` *Compute Relative Absolute Isometry Errors*

Description

Computes quantiles of pairwise relative absolute errors

$$\left|\alpha \hat{D}_{ij} - D_{ij}\right| / D_{ij}.$$

Usage

```
isometry.rel.abs.error(
  D.estimated,
  D.true,
  probs = c(0.5, 0.95),
  scale = TRUE,
  true.tol = sqrt(.Machine$double.eps)
)
```

Arguments

`D.estimated`	Estimated distance matrix or `dist` object.
`D.true`	Ground-truth distance matrix or `dist` object.
`probs`	Numeric vector of probabilities passed to `stats::quantile()`.
`scale`	Logical scalar. If `TRUE`, use `isometry.scale()` to optimally rescale estimated distances before computing the error.
`true.tol`	Non-negative tolerance. Pairs with true distance less than or equal to this value are excluded from relative metrics.

Value

Named numeric vector of relative absolute error quantiles.

Examples

```
D <- as.matrix(dist(1:4))
isometry.rel.abs.error(1.1 * D, D)
```

`isometry.rel.rms.error` *Compute Relative RMS Isometry Error*

Description

Computes the relative root-mean-square distance-matrix error after optional scalar calibration:

$$\left[\frac{\sum_{i<j}(\alpha\hat{D}_{ij} - D_{ij})^2}{\sum_{i<j} D_{ij}^2}\right]^{1/2}.$$

Usage

```
isometry.rel.rms.error(
  D.estimated,
  D.true,
  scale = TRUE,
  true.tol = sqrt(.Machine$double.eps)
)
```

Arguments

`D.estimated`	Estimated distance matrix or `dist` object.
`D.true`	Ground-truth distance matrix or `dist` object.
`scale`	Logical scalar. If `TRUE`, use `isometry.scale()` to optimally rescale estimated distances before computing the error.
`true.tol`	Non-negative tolerance. Pairs with true distance less than or equal to this value are excluded from relative metrics.

Value

Numeric scalar relative RMS error.

Examples

```
D <- as.matrix(dist(1:4))
isometry.rel.rms.error(2 * D, D)
```

`isometry.scale` *Compute the Optimal Isometry Calibration Scale*

Description

Computes the least-squares scalar α that aligns estimated distances to true distances:

$$\alpha^* = \frac{\sum_{i<j} \hat{D}_{ij} D_{ij}}{\sum_{i<j} \hat{D}_{ij}^2}.$$

Usage

```
isometry.scale(D.estimated, D.true, true.tol = sqrt(.Machine$double.eps))
```

Arguments

`D.estimated`	Estimated distance matrix or `dist` object.
`D.true`	Ground-truth distance matrix or `dist` object.
`true.tol`	Non-negative tolerance. Pairs with true distance less than or equal to this value are excluded from relative metrics.

Value

Numeric scalar calibration scale.

Examples

```
D <- as.matrix(dist(1:4))
isometry.scale(2 * D, D)
```

`jensen.shannon.divergence` *Jensen-Shannon Divergence*

Description

Jensen-Shannon Divergence

Usage

```
jensen.shannon.divergence(p, q)
```

Arguments

p	Numeric probability vector.
q	Numeric probability vector.

Value

Jensen-Shannon divergence using log base 2.

---

join.graphs *Join Two Adjacency-List Graphs*

Description

Join Two Adjacency-List Graphs

Usage

```
join.graphs(graph1, graph2, i1, i2)
```

Arguments

graph1	First adjacency list.
graph2	Second adjacency list.
i1	Vertex in graph1 to connect.
i2	Vertex in graph2 to connect.

Value

A joined adjacency list using 1-based vertex indices.

---

load.graph.data *Load Graph Data from RDA Files*

Description

Loads pruned graph data from a series of RDA files for different k values and combines them into a list structure containing adjacency and distance lists.

Usage

```
load.graph.data(
  k.values,
  prefix,
  suffix = "_NN_graph.rda",
  graph.object.name = "S.graph",
  verbose = TRUE
)
```

Arguments

k.values	Numeric vector of k values to process. Must be positive integers.
prefix	Character string specifying the path and prefix of the RDA files.
suffix	Character string specifying the suffix of the RDA files (default: "_NN_graph.rda").
graph.object.name	Character string specifying the name of the graph object stored in the RDA files (default: "S.graph").
verbose	Logical indicating whether to show progress messages (default: TRUE).

Details

The function expects RDA files to be named according to the pattern: `paste0(prefix, k, suffix)` for each k value.

Each RDA file should contain an object with the name specified in `graph.object.name`, which must have components `pruned_adj_list` and `pruned_dist_list`.

Value

A list containing two components:

adj.list	Named list of adjacency lists, one for each k value
dist.list	Named list of distance lists, one for each k value

Examples

```
prefix <- file.path(tempdir(), "dgraphs-k")
S.graph <- list(
  pruned_adj_list = list(c(2L), c(1L)),
  pruned_dist_list = list(1, 1)
)
save(S.graph, file = paste0(prefix, 2, ".rda"))
graph.data <- load.graph.data(
  2, prefix = prefix, suffix = ".rda", verbose = FALSE
)
unlink(paste0(prefix, 2, ".rda"))
names(graph.data$adj.list)
```

---

minh.limit *Find the Minimum Hop Limit for Path Existence*

Description

Find the Minimum Hop Limit for Path Existence

Usage

```
minh.limit(x, from, to)
```

Arguments

x	A path.graph.series object.
from	Source vertex index.
to	Target vertex index.

Value

Minimum hop limit where the path exists, or NULL.

---

nerve.graph *Construct the Nerve Graph of a Cover*

Description

Construct the Nerve Graph of a Cover

Usage

```
nerve.graph(covering.list, n.cores = 1)
```

Arguments

covering.list	List of integer vectors, one for each set in the cover.
n.cores	Number of parallel workers. Use 1 for serial execution.

Value

A list with adjacency.list, weights.list, and adjacency.matrix.

overlap.distribution.plot *Plot the distribution of overlap values*

Description

Plot the distribution of overlap values

Usage

```
overlap.distribution.plot(x, radius_idx = NULL)
```

Arguments

x	A vertex_geodesic_stats object
radius_idx	Index of the radius to display. If NULL, uses the radius with the most composite geodesics.

Value

Invisibly returns NULL

path.dist *Normalized Cumulative Distance Along a Vertex Path*

Description

Normalized Cumulative Distance Along a Vertex Path

Usage

```
path.dist(s, V, edge.col = "gray")
```

Arguments

s	Sequence of vertex indices.
V	Vertex coordinate matrix.
edge.col	Legacy argument retained for compatibility.

Value

Numeric vector of cumulative path distances normalized to end at 1.

path.length *Compute Euclidean Path Length*

Description

Compute Euclidean Path Length

Usage

```
path.length(X)
```

Arguments

X	Numeric matrix whose rows are consecutive path points.

Value

Total Euclidean length of the path.

plot.build_iknn_graphs_and_selectk *Plot method for build_iknn_graphs_and_selectk*

Description

Plot method for build_iknn_graphs_and_selectk

Usage

```
## S3 method for class 'build_iknn_graphs_and_selectk'
plot(
  x,
  which = c("connect", "edit", "mixing"),
  connect.args = list(),
  edit.args = list(),
  mixing.args = list(),
  par.args = list(),
  ...
)
```

Arguments

x	Object from build.iknn.graphs.and.selectk().
which	Character vector selecting panels among "connect", "edit", and "mixing".
connect.args, edit.args, mixing.args	Named lists forwarded to panel plots.
par.args	Named list forwarded to graphics::par().
...	Additional arguments, currently ignored.

Value

Invisibly returns x, unchanged, after producing the requested panels. If which is empty, invisibly returns NULL without plotting.

---

plot.cst_graph_mixing_stats

Plot method for cst_graph_mixing_stats

Description

Plot method for cst_graph_mixing_stats

Usage

```
## S3 method for class 'cst_graph_mixing_stats'
plot(
  x,
 which = c("null.homophily", "null.assortativity", "mixing.matrix", "conductance"),
  ...
)
```

Arguments

x	Object from cst.graph.mixing.stats().
which	Character vector specifying panels to plot.
...	Base graphics arguments.

Value

Invisibly returns NULL after producing the requested diagnostic panels.

plot.geodesic_stats *Plot Geodesic Statistics*

Description

Creates visualizations of geodesic statistics to help understand the relationship between radius and number of geodesics.

Usage

```
## S3 method for class 'geodesic_stats'
plot(
  x,
  plot.type = c("summary", "heatmap", "vertex", "all"),
  selected.vertices = NULL,
  max.vertices = 10,
  ...
)
```

Arguments

x	List. Output from compute.geodesic.stats().
plot.type	Character. Type of plot to generate: "summary" (default), "heatmap", "vertex", or "all".
selected.vertices	Integer vector. Specific vertices to highlight in "vertex" plots. If NULL, a random sample will be used.
max.vertices	Integer. Maximum number of vertices to show in vertex plot.
...	Additional arguments passed to summary

Value

Invisibly returns NULL.

Examples

```
stats <- list(
  radii = c(0.2, 0.4),
  geodesic_rays = matrix(c(2, 3, 3, 4), nrow = 2, byrow = TRUE),
  composite_geodesics = matrix(c(1, 1, 1, 2), nrow = 2, byrow = TRUE),
  grid_vertices = c(5L, 9L),
  summary = data.frame(
    radius = c(0.2, 0.4),
    avg_rays = c(2.5, 3.5),
    max_rays = c(3, 4),
    composite_ratio = c(0.4, 0.5),
    avg_overlap_median = c(0.35, 0.45)
```

```
  )
)
class(stats) <- c("geodesic_stats", "list")

plot(stats, plot.type = "summary")
plot(stats, plot.type = "vertex", selected.vertices = 5)
```

`plot.IkNNgraphs` *Plot Diagnostics for Intersection k-NN Graph Analysis*

Description

Plot Diagnostics for Intersection k-NN Graph Analysis

Usage

```
## S3 method for class 'IkNNgraphs'
plot(
  x,
  type = "diag",
  diags = c("edist", "edge", "deg"),
  with.pwlm = TRUE,
  with.lmin = FALSE,
  breakpoint.col = "blue",
  lmin.col = "gray",
  k = NA,
  mar = c(2.5, 2.5, 0.5, 0.5),
  mgp = c(2.5, 0.5, 0),
  tcl = -0.3,
  xline = 2.4,
  yline = 3.15,
  ...
)
```

Arguments

`x`	A list with `k.values`, `edit.distances`, `n.edges.in.pruned.graph`, and `js.div`.
`type`	Character string. Only `"diag"` is currently supported.
`diags`	Diagnostic panels. The supported combination is `c("edist", "edge", "deg")`.
`with.pwlm`	Logical. If `TRUE`, overlays smoothed trend fits.
`with.lmin`	Logical. If `TRUE`, shows local-minimum vertical lines.
`breakpoint.col`	Color for breakpoint vertical lines.
`lmin.col`	Color for local-minima vertical lines.
`k`	Optional k to highlight. Reserved for legacy compatibility.
`mar`, `mgp`, `tcl`, `xline`, `yline`	Base graphics controls.
`...`	Additional arguments passed to plot.

Value

Invisibly returns TRUE.

---

plot.iknn_stability_metrics *Plot Method for IkNN Stability Metrics*

Description

Plot Method for IkNN Stability Metrics

Usage

```
## S3 method for class 'iknn_stability_metrics'
plot(x, ...)
```

Arguments

x	An object returned by compute.stability.metrics().
...	Passed to plot.IkNNgraphs().

Value

Invisibly returns TRUE after producing the stability diagnostic plot.

---

plot.vertex_geodesic_stats *Plot method for vertex_geodesic_stats objects*

Description

Plot method for vertex_geodesic_stats objects

Usage

```
## S3 method for class 'vertex_geodesic_stats'
plot(x, ...)
```

Arguments

x	A vertex_geodesic_stats object
...	Additional arguments passed to plot

Value

Invisibly returns NULL

Examples

```
x <- data.frame(
  radius = c(0.2, 0.4, 0.6),
  rays = c(3, 5, 6),
  composite_geodesics = c(1, 2, 3),
  overlap_min = c(0.10, 0.15, 0.20),
  overlap_p05 = c(0.15, 0.20, 0.25),
  overlap_p25 = c(0.20, 0.25, 0.30),
  overlap_median = c(0.30, 0.40, 0.50),
  overlap_p75 = c(0.40, 0.50, 0.60),
  overlap_p95 = c(0.55, 0.65, 0.75),
  overlap_max = c(0.60, 0.70, 0.80),
  potential_pairs = c(3, 10, 15),
  composite_ratio = c(1/3, 0.2, 0.2),
  overlap_ratio = c(0.30, 0.40, 0.50)
)
class(x) <- c("vertex_geodesic_stats", "data.frame")
attr(x, "vertex") <- 7L

plot(x)
```

---

plot2D.colored.graph *Plot a Graph with Colored Vertices*

Description

Plot a Graph with Colored Vertices

Usage

```
plot2D.colored.graph(
  embedding,
  adj.list,
  vertex.colors,
  vertex.size = 1,
  edge.alpha = 0.2,
  color.palette = NULL,
  main = "",
  add.legend = TRUE
)
```

Arguments

`embedding`	Numeric `n x 2` matrix of vertex coordinates.
`adj.list`	Graph adjacency list.
`vertex.colors`	Numeric color value for each vertex.

vertex.size	Base graphics point size.
edge.alpha	Edge alpha in [0, 1].
color.palette	Optional vector of colors.
main	Plot title.
add.legend	Logical; add color scale legend.

Value

Invisibly returns NULL; produces a plot as a side effect.

---

print.build_iknn_graphs_and_selectk *Print method for build_iknn_graphs_and_selectk*

Description

Print method for build_iknn_graphs_and_selectk

Usage

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

Arguments

x	Object from build.iknn.graphs.and.selectk().
...	Unused.

Value

Invisibly returns x, unchanged, after printing its selected neighborhood sizes and trimming status.

---

print.geodesic_stats *Print method for geodesic_stats objects*

Description

Prints a brief overview of the geodesic statistics.

Usage

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

Arguments

x	A geodesic_stats object from compute.geodesic.stats().
...	Additional arguments passed to print.

Value

The object invisibly.

---

print.knn.outliers *Print Method for knn.outliers Objects*

Description

Prints a summary of the outlier detection results from `remove.knn.outliers`.

Usage

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

Arguments

x	An object of class "knn.outliers" as returned by `remove.knn.outliers`.
...	Additional arguments passed to `print` methods.

Value

Invisibly returns the input object.

Examples

```
knn.res <- list(
  S.q = matrix(c(0, 0,
                 1, 1,
                 2, 2), ncol = 2, byrow = TRUE),
  y.q = c(1.0, 1.5, 2.0),
  nn.d = matrix(c(0.2, 0.4,
                  0.3, 0.5,
                  1.4, 1.8,
                  0.25, 0.45),
                ncol = 2, byrow = TRUE),
  d.thld = 1,
  idx = c(TRUE, TRUE, FALSE, TRUE),
  n.outliers = 1L,
  method = "qf"
)
class(knn.res) <- "knn.outliers"

knn.res
```

print.maximal_packing *Print Method for Maximal Packing Results*

Description

Print Method for Maximal Packing Results

Usage

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

Arguments

x	An object of class "maximal_packing"
...	Additional arguments (currently ignored)

Value

Invisible copy of x

Examples

```
x <- list(
  adj_list = list(c(2L), c(1L, 3L), c(2L, 4L), c(3L)),
  graph_diameter = 3,
  max_packing_radius = 1.5,
  grid_vertices = c(1L, 4L)
)
class(x) <- "maximal_packing"

x
```

print.mknn_graph *Print Method for mknn_graph Objects*

Description

Prints a concise summary of a mutual k-nearest neighbor graph object.

Usage

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

Arguments

x	An object of class "mknn_graph".
...	Additional arguments (currently unused).

Value

Invisibly returns the input object.

Examples

```
graph <- list(
  n_vertices = 3L,
  n_edges = 2L,
  k = 2L,
  adj_list = list(c(2L), c(1L, 3L), c(2L))
)
class(graph) <- c("mknn_graph", "list")

print(graph)
```

---

print.mknn_graphs *Print Method for mknn_graphs Objects*

Description

Prints a concise summary of a collection of mutual k-nearest neighbor graphs.

Usage

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

Arguments

x	An object of class "mknn_graphs".
...	Additional arguments (currently unused).

Value

Invisibly returns the input object.

Examples

```
graphs <- list(
  k_statistics = data.frame(
    k = 2:4,
    n_edges_pruned = c(2L, 3L, 3L)
  )
)
attr(graphs, "kmin") <- 2L
attr(graphs, "kmax") <- 4L
attr(graphs, "n_vertices") <- 3L
class(graphs) <- "mknn_graphs"

print(graphs)
```

---

print.mst_completion_graph

Print Method for MST Completion Graph Objects

Description

Displays a concise summary of an MST completion graph object, including basic graph statistics and parameter information.

Usage

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

Arguments

x	An object of class mst_completion_graph, as returned by create.cmst.graph.
...	Additional arguments (currently ignored).

Details

This method provides a human-readable overview of the graph structure without displaying the full adjacency lists. For detailed inspection of graph components, use direct indexing (e.g., x$mst_adj_list) or the summary method.

Value

Invisibly returns the input object x.

See Also

summary.mst_completion_graph for more detailed summaries, create.cmst.graph for creating MST completion graphs

Examples

```
graph <- list(
  mst_adj_list = list(c(2L), c(1L, 3L), c(2L)),
  mst_weight_list = list(1.1, c(1.1, 0.9), 0.9),
  cmst_adj_list = list(c(2L, 3L), c(1L, 3L), c(1L, 2L)),
  cmst_weight_list = list(c(1.1, 1.4), c(1.1, 0.9), c(1.4, 0.9)),
  mst_edge_weights = c(1.1, 0.9),
  cmst_distance_threshold = 1.08
)
attr(graph, "q_thld") <- 0.8
attr(graph, "cmst_distance_threshold") <- graph$cmst_distance_threshold
class(graph) <- c("mst_completion_graph", "list")

print(graph)
```

---

print.packing_validation

Print Method for Packing Validation Results

Description

Print Method for Packing Validation Results

Usage

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

Arguments

x	An object of class "packing_validation"
...	Additional arguments (currently ignored)

Value

Invisible copy of x

Examples

```
x <- list(
  valid = TRUE,
  is.maximal = TRUE,
  min.packing.distance = 2,
  max.coverage.distance = 1,
  violations = NULL,
  potential.additions = NULL
)
class(x) <- "packing_validation"
```

x

---

print.summary.knn.outliers
Print Method for summary.knn.outliers Objects

Description

Prints the summary of outlier detection results.

Usage

```
## S3 method for class 'summary.knn.outliers'
print(x, ...)
```

Arguments

x	An object of class "summary.knn.outliers".
...	Additional arguments passed to print methods.

Value

Invisibly returns the input object.

Examples

```
knn.res <- list(
  S.q = matrix(c(0, 0,
                 1, 1,
                 2, 2), ncol = 2, byrow = TRUE),
  y.q = c(1.0, 1.5, 2.0),
  nn.d = matrix(c(0.2, 0.4,
                  0.3, 0.5,
                  1.4, 1.8,
                  0.25, 0.45),
                ncol = 2, byrow = TRUE),
  d.thld = 1,
  idx = c(TRUE, TRUE, FALSE, TRUE),
  n.outliers = 1L,
  method = "qf"
)
class(knn.res) <- "knn.outliers"

summary(knn.res)
```

`print.summary.mst_completion_graph` *Print Summary of MST Completion Graph*

Description

Formats and displays the summary information for an MST completion graph in a readable format.

Usage

```
## S3 method for class 'summary.mst_completion_graph'
print(x, digits = 3L, ...)
```

Arguments

`x`	An object of class `summary.mst_completion_graph`, as returned by `summary.mst_completion_graph`.
`digits`	Integer; number of significant digits for numeric output. Default is 3.
`...`	Additional arguments (currently ignored).

Value

Invisibly returns the input summary object.

Examples

```
graph <- list(
  mst_adj_list = list(c(2L), c(1L, 3L), c(2L)),
  mst_weight_list = list(1.1, c(1.1, 0.9), 0.9),
  cmst_adj_list = list(c(2L, 3L), c(1L, 3L), c(1L, 2L)),
  cmst_weight_list = list(c(1.1, 1.4), c(1.1, 0.9), c(1.4, 0.9)),
  mst_edge_weights = c(1.1, 0.9),
  cmst_distance_threshold = 1.08
)
attr(graph, "q_thld") <- 0.8
attr(graph, "cmst_distance_threshold") <- graph$cmst_distance_threshold
class(graph) <- c("mst_completion_graph", "list")

graph_summary <- summary(graph)
print(graph_summary, digits = 4)
```

`remove.knn.outliers` *Remove Outliers from a State Space Using k-Nearest Neighbors*

Description

Identifies and removes outliers from a multivariate state space based on k-nearest neighbor distances. This function implements several strategies for outlier detection, all based on the principle that outliers tend to be isolated from the main data clusters and thus have larger distances to their nearest neighbors.

Usage

```
remove.knn.outliers(
  S,
  y = NULL,
  p = 0.98,
  dist.factor = 100,
  K = 30,
  method = "diff.dist.factor"
)
```

Arguments

`S`	A numeric matrix or data frame representing the state space, where each row is an observation and each column is a dimension or feature. Must contain at least `K+1` observations.
`y`	An optional numeric vector containing values of a variable defined over the state space. If provided, must have length equal to `nrow(S)`. The function will filter this vector to match the filtered state space.
`p`	A numeric value between 0 and 1 (exclusive) indicating the percentile threshold for outlier removal. The default value of 0.98 removes points whose distance to the nearest neighbor exceeds the 98th percentile of all nearest neighbor distances.
`dist.factor`	A positive numeric value used as a threshold factor in certain outlier detection methods. In "diff.dist.factor", points with relative jumps greater than this factor are considered outliers. Default is 100.
`K`	A positive integer specifying the number of nearest neighbors to compute for each point. Must be less than the number of observations in `S`. Default is 30.
`method`	A character string specifying the outlier detection method to use when K > 1. Must be one of "dist.factor", "diff.dist.factor" (default), or "default" for the unnamed method.

Details

The function determines outliers by analyzing the distances between points and their K nearest neighbors. Several detection methods are available:

When K = 1: Points are flagged as outliers if their distance to the nearest neighbor exceeds the p-th percentile threshold of all nearest neighbor distances.

"dist.factor": Points are considered outliers if the ratio of the K-th nearest neighbor distance to the 1st nearest neighbor distance exceeds `dist.factor`.

"diff.dist.factor" (default): For each point, finds the maximum jump in distance between consecutive neighbors, normalizes by the median jump across all points, and flags points as outliers if their relative jump exceeds `dist.factor`. This method is particularly robust as it adapts to the overall distribution of distances.

Default unnamed method: Iteratively checks if the first nearest neighbor distance exceeds the threshold or if any consecutive difference between neighbor distances exceeds the threshold.

Value

A list of class "knn.outliers" containing the following components:

S.q The filtered state space matrix with outliers removed.

y.q The filtered dependent variable (if `y` was provided), otherwise `NULL`.

nn.d A matrix of dimensions `nrow(S)` x `K` containing distances to the K nearest neighbors for each point.

d.thld The distance threshold used for outlier detection.

idx A logical vector indicating which points were kept (`TRUE`) and which were identified as outliers (`FALSE`).

n.outliers The number of outliers detected.

method The outlier detection method used.

See Also

`get.knn` for the k-nearest neighbor calculation, `quantile` for percentile calculations

Examples

```
# Create a sample dataset with outliers
set.seed(123)
n_normal <- 120
n_outliers <- 6

# Generate normal data
normal_data <- matrix(rnorm(n_normal * 2), ncol = 2)

# Generate outliers far from the main cluster
outliers <- cbind(rnorm(n_outliers, mean = 10, sd = 0.5),
                  rnorm(n_outliers, mean = 10, sd = 0.5))

# Combine data
S <- rbind(normal_data, outliers)
y <- c(rnorm(n_normal), rnorm(n_outliers, mean = 5))

# Remove outliers using the default method
```

```
result <- remove.knn.outliers(S, y, K = 10)

# Print summary
cat("Number of outliers detected:", result$n.outliers, "\n")
cat("Percentage of data retained:",
    round(100 * nrow(result$S.q) / nrow(S), 2), "%\n")

# Use a different method with more conservative threshold
result2 <- remove.knn.outliers(S, y, p = 0.95, method = "dist.factor",
                               dist.factor = 5, K = 10)

# Plot the original and filtered state space
oldpar <- par(mfrow = c(1, 2))
plot(S, col = "gray", main = "Original State Space",
     xlab = "Dimension 1", ylab = "Dimension 2")
points(S[!result$idx, ], col = "red", pch = 16, cex = 1.2)

plot(result$S.q, col = "blue", pch = 16,
     main = "Filtered State Space",
     xlab = "Dimension 1", ylab = "Dimension 2")
legend("topright", legend = c("Retained", "Removed"),
       col = c("blue", "red"), pch = 16)
par(oldpar)
```

`rm.self.loops` *Remove Self-Loops from an Adjacency List*

Description

Remove Self-Loops from an Adjacency List

Usage

```
rm.self.loops(adj.list)
```

Arguments

`adj.list`	Adjacency list.

Value

The adjacency list with entries equal to their vertex index removed.

shortest.path *Computes Shortest Path Distances for Selected Vertices*

Description

Computes Shortest Path Distances for Selected Vertices

Usage

```
shortest.path(graph, edge.lengths, vertices)
```

Arguments

graph	A graph adjacency list using 1-based vertex indices.
edge.lengths	Edge-length list matching graph.
vertices	Integer vector of vertices for which to compute distances.

Value

A numeric matrix of shortest-path distances.

subdivide.path *Subdivide a Path into Arc-Length Spaced Points*

Description

Subdivide a Path into Arc-Length Spaced Points

Usage

```
subdivide.path(path, n.subdivision.pts)
```

Arguments

path	Matrix of consecutive path points.
n.subdivision.pts	Number of output points.

Value

Matrix of subdivided path coordinates.

summarize.isometry.deviation
Summarize Deviation from Isometry

Description

Computes the standard benchmark summary used to compare graph geodesic distances with reference geodesic distances: optimal scalar calibration, relative RMS error, relative absolute error quantiles, multiplicative distortion quantiles, and distance correlations.

Usage

```
summarize.isometry.deviation(
  D.estimated,
  D.true,
  scale = TRUE,
  true.tol = sqrt(.Machine$double.eps)
)
```

Arguments

D.estimated	Estimated distance matrix or dist object.
D.true	Ground-truth distance matrix or dist object.
scale	Logical scalar. If TRUE, use isometry.scale() to optimally rescale estimated distances before computing the error.
true.tol	Non-negative tolerance. Pairs with true distance less than or equal to this value are excluded from relative metrics.

Value

A one-row data frame.

Examples

```
D <- as.matrix(dist(1:4))
summarize.isometry.deviation(1.1 * D, D)
```

summary.geodesic_stats *Summary method for geodesic_stats objects*

Description

Provides a summary table of geodesic statistics with percentages and counts for different metric types.

Usage

```
## S3 method for class 'geodesic_stats'
summary(object, type = c("rays", "composite", "overlap"), ...)
```

Arguments

object	A geodesic_stats object from compute.geodesic.stats().
type	Character. Type of summary to generate: "rays" (default), "composite", or "overlap".
...	Additional arguments passed to summary.

Value

A matrix with row for each radius and columns for each unique count value. Each cell contains a string with "percent (count)" format.

Examples

```
stats <- list(
  radii = c(0.2, 0.4),
  geodesic_rays = matrix(c(2, 3, 3, 4), nrow = 2, byrow = TRUE),
  composite_geodesics = matrix(c(1, 1, 1, 2), nrow = 2, byrow = TRUE),
  path_overlap = list(
    min = matrix(c(0.1, 0.2, 0.2, 0.3), nrow = 2, byrow = TRUE),
    p05 = matrix(c(0.1, 0.2, 0.2, 0.3), nrow = 2, byrow = TRUE),
    p25 = matrix(c(0.2, 0.3, 0.3, 0.4), nrow = 2, byrow = TRUE),
    median = matrix(c(0.3, 0.4, 0.4, 0.5), nrow = 2, byrow = TRUE),
    p75 = matrix(c(0.4, 0.5, 0.5, 0.6), nrow = 2, byrow = TRUE),
    p95 = matrix(c(0.5, 0.6, 0.6, 0.7), nrow = 2, byrow = TRUE),
    max = matrix(c(0.6, 0.7, 0.7, 0.8), nrow = 2, byrow = TRUE)
  )
)
class(stats) <- c("geodesic_stats", "list")

summary(stats)
summary(stats, type = "composite")
summary(stats, type = "overlap")
```

summary.IkNN *Summarize IkNN Graph Object*

Description

Prints a summary of an IkNN graph object, including the number of vertices, edges, and pruning statistics.

Usage

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

Arguments

object	An object of class "IkNN", typically output from create.single.iknn.graph()
...	Additional arguments passed to summary().

Value

Invisibly returns NULL while printing summary information to the console

Examples

```
graph <- list(
  pruned_adj_list = list(c(2L, 3L), c(1L, 3L), c(1L, 2L)),
  n_edges = 3L,
  n_pruned_edges = 2L,
  n_removed_edges = 1L,
  edge_reduction_ratio = 1 / 3
)
class(graph) <- c("IkNN", "list")

summary(graph)
```

summary.iknn_graphs *Summarize an iknn_graphs Object*

Description

Provides a detailed summary of an iknn_graphs object created by the create.iknn.graphs() function. The summary includes statistics for each intersection kNN graph in the sequence, displaying information about the connectivity and structure of the graphs for different k values.

Usage

```
## S3 method for class 'iknn_graphs'
summary(object, use.isize.pruned = FALSE, ...)
```

Arguments

object	An object of class 'iknn_graphs', typically the output of create.iknn.graphs().
use.isize.pruned	Logical. If TRUE, computes and displays statistics for the intersection-size pruned graphs (isize_pruned_graphs). If FALSE (default), computes statistics for the geometrically pruned graphs (geom_pruned_graphs).
...	Additional arguments passed to or from other methods (not currently used).

Details

The summary function extracts and presents key statistics about the structure of each intersection kNN graph in the iknn_graphs object. All graphs share the same number of vertices (equal to the number of rows in the input data matrix), but they differ in the number of edges due to varying k values and the pruning methods applied during graph creation.

The function displays general information about the graph sequence, including the number of vertices, the range of k values, and the pruning parameters used. It then presents a tabular summary of statistics for each individual graph, showing how the graph structure changes as k increases.

For each intersection kNN graph, the following metrics are calculated:

- Number of edges: Total number of connections in the graph
- Mean degree: Average number of connections per vertex
- Min/Max degree: Range of vertex connectivity
- Density: Proportion of potential connections that are actually present
- Sparsity: 1 - density, measuring the proportion of missing connections

Value

Invisibly returns a data frame containing statistics for each graph. The data frame has the following columns:

idx	The index of the given k value
k	The k value for the intersection kNN graph
n_ccomp	Number of connected components of the graph
edges	Number of edges in the graph
mean_degree	Average number of connections per vertex
min_degree	Minimum vertex degree (least connected vertex)
max_degree	Maximum vertex degree (most connected vertex)
sparsity	Graph sparsity, calculated as 1 - density. It measures how many (proportion) potential connections are missing.

See Also

create.iknn.graphs for creating intersection kNN graphs.

Examples

```
# Create sample data
set.seed(123)
x <- matrix(rnorm(1000), ncol = 5)

# Generate intersection kNN graphs
iknn.res <- create.iknn.graphs(
  x,
  kmin = 3,
  kmax = 10,
  n.cores = 1,
  with.isize.pruning = TRUE
)

# Summarize the geometrically pruned graphs
summary(iknn.res)

# Summarize the intersection-size pruned graphs
summary(iknn.res, use.isize.pruned = TRUE)
```

---

summary.knn.outliers *Summary Method for knn.outliers Objects*

Description

Provides a detailed summary of the outlier detection results from remove.knn.outliers.

Usage

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

Arguments

object	An object of class "knn.outliers" as returned by remove.knn.outliers.
...	Additional arguments (currently unused).

Value

A list of class "summary.knn.outliers" containing summary statistics.

Examples

```
knn.res <- list(
  S.q = matrix(c(0, 0,
                 1, 1,
                 2, 2), ncol = 2, byrow = TRUE),
  y.q = c(1.0, 1.5, 2.0),
  nn.d = matrix(c(0.2, 0.4,
                  0.3, 0.5,
                  1.4, 1.8,
                  0.25, 0.45),
                ncol = 2, byrow = TRUE),
  d.thld = 1,
  idx = c(TRUE, TRUE, FALSE, TRUE),
  n.outliers = 1L,
  method = "qf"
)
class(knn.res) <- "knn.outliers"

summary(knn.res)
```

---

summary.mknn_graphs *Summary Method for mknn_graphs Objects*

Description

Provides a comprehensive summary of mutual k-nearest neighbor graphs created by `create.mknn.graphs`, including structural statistics and connectivity information for each k value.

Usage

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

Arguments

`object`	An object of class "mknn_graphs", typically output from `create.mknn.graphs`.
`...`	Additional arguments (currently unused).

Details

The summary provides insights into how graph structure changes with k:

- **Connected Components**: Indicates graph fragmentation. A value of 1 means the graph is fully connected.
- **Degree Statistics**: Shows the distribution of vertex connectivity. Large differences between min and max degree may indicate hub vertices.
- **Density/Sparsity**: Measures how many of the possible edges are present. MkNN graphs are typically very sparse.

For pruned graphs, the statistics reflect the structure after geometric pruning has been applied.

Value

Invisibly returns a data frame containing graph statistics with columns:

idx Index of the k value (1 to number of graphs)

k The k value used for the graph

n_components Number of connected components

n_edges Number of edges in the graph

mean_degree Average vertex degree

min_degree Minimum vertex degree

max_degree Maximum vertex degree

density Graph density (proportion of possible edges present)

sparsity Graph sparsity (1 - density)

Examples

```
mknn_result <- list(
  pruned_graphs = list(
    list(adj_list = list(c(2L), c(1L, 3L), c(2L))),
    list(adj_list = list(c(2L, 3L), c(1L, 3L), c(1L, 2L)))
  ),
  k_statistics = data.frame(
    k = 2:3,
    n_edges = c(2L, 3L),
    n_edges_pruned = c(2L, 3L),
    n_removed = c(0L, 0L),
    reduction_ratio = c(0, 0)
  )
)
attr(mknn_result, "kmin") <- 2L
attr(mknn_result, "kmax") <- 3L
attr(mknn_result, "n_vertices") <- 3L
attr(mknn_result, "max.path.edge.ratio.thld") <- 1.2
class(mknn_result) <- "mknn_graphs"

stats <- summary(mknn_result)
plot(stats$k, stats$mean_degree, type = "b",
     xlab = "k", ylab = "Mean Degree")
```

---

summary.mst_completion_graph

Summary Method for MST Completion Graph Objects

Description

Computes and returns a comprehensive summary of an MST completion graph, including graph statistics, edge weight distributions, and PCA information if applicable.

Usage

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

Arguments

object	An object of class mst_completion_graph.
...	Additional arguments (currently ignored).

Value

An object of class summary.mst_completion_graph containing:

n_vertices Number of vertices in the graph

n_mst_edges Number of edges in the minimal spanning tree

n_cmst_edges Number of edges in the completed graph

q_thld Quantile threshold used for completion

edge_weight_summary Five-number summary of MST edge weights

completion_threshold The actual distance threshold used

pca_applied Logical indicating if PCA was applied

pca_info PCA details (if applicable)

See Also

print.summary.mst_completion_graph for printing summaries, create.cmst.graph for creating graphs

Examples

```
graph <- list(
  mst_adj_list = list(c(2L), c(1L, 3L), c(2L)),
  mst_weight_list = list(1.1, c(1.1, 0.9), 0.9),
  cmst_adj_list = list(c(2L, 3L), c(1L, 3L), c(1L, 2L)),
  cmst_weight_list = list(c(1.1, 1.4), c(1.1, 0.9), c(1.4, 0.9)),
  mst_edge_weights = c(1.1, 0.9),
  cmst_distance_threshold = 1.08
)
attr(graph, "q_thld") <- 0.8
attr(graph, "cmst_distance_threshold") <- graph$cmst_distance_threshold
class(graph) <- c("mst_completion_graph", "list")

summary(graph)
```

summary.rknn_graphs *Summarize an rknn_graphs Object*

Description

Provides a compact table of graph characteristics for an "rknn_graphs" object created by create.rknn.graphs() or cpp.create.rknn.graphs(). The table reports final edge counts, component-repair counts, degree summaries, and density/sparsity for each adaptive-radius k.scale value.

Usage

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

Arguments

object	An object of class "rknn_graphs".
...	Additional arguments passed to or from other methods; currently unused.

Value

Invisibly returns a data frame with one row per graph and columns:

idx	Position of the graph in the returned graph sequence.
k	The adaptive-radius k.scale value.
n_vertices	Number of graph vertices.
n_ccomp	Number of connected components in the final graph.
n_ccomp_before_repair	Number of components before component-MST repair.
edges	Final number of undirected edges.
edges_before_pruning	Number of edges before geometric pruning.
edges_after_pruning	Number of edges after geometric pruning and before optional component repair.
mst_edges_added	Number of component-MST repair edges added.
mean_degree	Average vertex degree in the final graph.
min_degree	Minimum vertex degree.
median_degree	Median vertex degree.
max_degree	Maximum vertex degree.
max_degree_over_median	Ratio of maximum to median degree, or NA when the median degree is zero.
universal_vertices	Number of vertices adjacent to every other vertex.
density	Final graph density.
sparsity	One minus graph density.

See Also

create.rknn.graphs(), cpp.create.rknn.graphs()

Examples

```
X <- matrix(c(0, 1, 3, 4), ncol = 1)
graphs <- create.rknn.graphs(
  X,
  kmin = 1,
  kmax = 2,
  radius.search = "all.pairs",
  graph.detail = "minimal",
  prune.method = "none"
)
summary(graphs)
```

---

summary.vertex_geodesic_stats

Summary method for vertex_geodesic_stats objects

Description

Summary method for vertex_geodesic_stats objects

Usage

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

Arguments

object	A vertex_geodesic_stats object
...	Additional arguments passed to summary

Value

A summary of the vertex geodesic statistics

Examples

```
x <- data.frame(
  radius = c(0.2, 0.4, 0.6),
  rays = c(3, 5, 6),
  composite_geodesics = c(1, 2, 3),
  overlap_min = c(0.10, 0.15, 0.20),
  overlap_p05 = c(0.15, 0.20, 0.25),
  overlap_p25 = c(0.20, 0.25, 0.30),
```

```
  overlap_median = c(0.30, 0.40, 0.50),
  overlap_p75 = c(0.40, 0.50, 0.60),
  overlap_p95 = c(0.55, 0.65, 0.75),
  overlap_max = c(0.60, 0.70, 0.80),
  potential_pairs = c(3, 10, 15),
  composite_ratio = c(1/3, 0.2, 0.2),
  overlap_ratio = c(0.30, 0.40, 0.50)
)
class(x) <- c("vertex_geodesic_stats", "data.frame")
attr(x, "vertex") <- 7L

summary(x)
```

`validate.maximal.packing` *Validate a Maximal Packing*

Description

Validates whether a given vertex packing is maximal and correctly satisfies the distance constraints. A packing is valid if all vertices are separated by at least the specified radius, and is maximal if no additional vertices can be added without violating this constraint.

Usage

```
validate.maximal.packing(
  adj.list,
  weight.list,
  packing.vertices,
  max.packing.radius
)
```

Arguments

`adj.list`	A list where each element is a vector of adjacent vertex indices (1-based) for the corresponding vertex.
`weight.list`	A list where each element is a vector of edge weights corresponding to the adjacencies in `adj.list`.
`packing.vertices`	An integer vector of vertex indices (1-based) that form the packing to be validated.
`max.packing.radius`	A numeric value representing the minimum distance that should separate any two vertices in the packing.

Details

This function performs two key validations:

1. **Packing Property**: Verifies that all vertices in the packing are separated by at least `max.packing.radius`.
2. **Maximality**: Verifies that no additional vertex can be added to the packing without violating the packing property.

The function uses the `igraph` package to compute shortest path distances between vertices using Dijkstra's algorithm.

Value

A list with class "packing_validation" containing validation results:

`valid` Logical indicating whether the packing satisfies the distance constraint

`min.packing.distance` The minimum distance found between any two packing vertices

`max.coverage.distance` The maximum distance from any non-packing vertex to its nearest packing vertex

`violations` A data frame containing details of any violations found, or NULL if none. Contains columns: type, vertex1, vertex2, distance

`is.maximal` Logical indicating whether the packing is maximal (no vertices can be added)

`potential.additions` Integer vector of vertex indices that could potentially be added to the packing if it's not maximal, or NULL if maximal

Examples

```
adj.list <- list(c(2), c(1, 3), c(2, 4), c(3, 5), c(4))
weight.list <- list(c(1), c(1, 1), c(1, 1), c(1, 1), c(1))
packing <- c(1, 3, 5)
radius <- 2

result <- validate.maximal.packing(adj.list, weight.list, packing, radius)
result$valid
result$is.maximal
```

---

verify.maximal.packing

Verify Maximal Packing Created by create.maximal.packing

Description

A convenience function to verify the correctness of a maximal packing created using the `create.maximal.packing` function. This function validates both the packing property (minimum separation distance) and maximality (no vertices can be added).

Usage

```
verify.maximal.packing(packing.result, verbose = TRUE)
```

Arguments

packing.result	The result returned by create.maximal.packing, which must be an object of class "maximal_packing".
verbose	Logical indicating whether to print detailed validation results. Default is TRUE.

Details

This function takes the output of create.maximal.packing and verifies two key properties:

1. The packing vertices are all separated by at least max_packing_radius
2. The packing is maximal (no more vertices can be added without violating the distance constraint)

```
When \code{verbose = TRUE}, the function prints:
\itemize{
  \item Graph diameter
  \item Packing radius used
  \item Number of vertices in the packing
  \item Minimum distance between packing vertices
  \item Maximum coverage distance
  \item Validity and maximality status
  \item Any violations or potential additions (if applicable)
}
```

Value

A logical value: TRUE if the packing is both valid (satisfies the distance constraint) and maximal (no vertices can be added), FALSE otherwise.

See Also

create.maximal.packing, validate.maximal.packing

Examples

```
adj.list <- list(c(2), c(1, 3), c(2, 4), c(3, 5), c(4))
weight.list <- list(c(1), c(1, 1), c(1, 1), c(1, 1), c(1))
result <- create.maximal.packing(adj.list, weight.list, grid.size = 2)

verify.maximal.packing(result, verbose = FALSE)
```

vertices *Extract Vertices from a Graph Result*

Description

Extract Vertices from a Graph Result

Usage

```
vertices(object, ...)
```

Arguments

object	Object from which vertices should be extracted.
...	Additional arguments passed to methods.

Value

Object-specific vertex indices.

wgraph.prune.long.edges *Prune Long Edges in a Weighted Graph*

Description

Prune Long Edges in a Weighted Graph

Usage

```
wgraph.prune.long.edges(
  graph,
  edge.lengths,
  alt.path.len.ratio.thld,
  use.total.length.constraint = TRUE,
  verbose = FALSE
)
```

Arguments

graph	A graph adjacency list.
edge.lengths	Edge-length list matching graph.
alt.path.len.ratio.thld	Alternative-path threshold.
use.total.length.constraint	If TRUE, compare total alternative path length with the original edge. Otherwise require every edge on the alternative path to be shorter than the original edge times the threshold.
verbose	Logical progress flag.

Value

A list with pruned adjacency and edge-length lists.

Index