Package ‘flownet’

January 27, 2026

Type Package

Title Transport Modeling: Network Processing, Route Enumeration, and Traffic Assignment

Version 0.1.1

Description High-performance tools for transport modeling - network processing, route enumeration, and traffic assignment in R. The package implements the Path-Sized Logit model for traffic assignment - Ben-Akiva and Bierlaire (1999) <doi:10.1007/978-1-4615-5203-1_2> -
an efficient route enumeration algorithm, and provides powerful utility functions for (multimodal) network generation, consolidation/contraction, and/or simplification. The user is expected to provide
a transport network (either a graph or collection of linestrings) and an origin-destination (OD) matrix of trade/traffic flows. Maintained by transport consultants at CPCS (cpcs.ca).

URL `https://sebkrantz.github.io/flownet/`,
`https://github.com/SebKrantz/flownet`

BugReports `https://github.com/SebKrantz/flownet/issues`

License GPL-3

Encoding UTF-8

Depends R (>= 4.1)

Imports collapse (>= 2.1.5), kit (>= 0.0.5), sf (>= 1.0.0), igraph (>= 2.1.4), geodist (>= 0.1.1), leaderCluster (>= 1.5.0), mirai (>= 2.5.2), progress (>= 1.2.3)

Suggests fastverse (>= 0.3.4), mapview (>= 2.11.2), tmap (>= 4.0), testthat (>= 3.0.0), knitr, rmarkdown

LazyData true

NeedsCompilation yes

RoxygenNote 7.3.2

VignetteBuilder knitr

Author Sebastian Krantz [aut, cre],
Kamol Roy [ctb]

Maintainer Sebastian Krantz <sebastian.krantz@graduateinstitute.ch>

Repository CRAN

Date/Publication 2026-01-27 21:40:02 UTC

Contents

`flownet-package` *Efficient Transport Modeling*

Description

flownet provides efficient tools for transportation modeling in R, supporting network processing, route enumeration, and traffic assignment tasks. It implements the path-sized logit (PSL) model for traffic assignment and provides powerful utilities for network processing/preparation.

Network Processing

`linestrings_to_graph()` — Convert LINESTRING geometries to graph
`create_undirected_graph()` — Convert directed graph to undirected
`consolidate_graph()` — Consolidate graph by removing intermediate nodes
`simplify_network()` — Simplify network graph

Traffic Assignment

`run_assignment()` — Run traffic assignment using path-sized logit model

Graph Utilities

`normalize_graph()` — Normalize node IDs to consecutive integers
`nodes_from_graph()` — Extract unique nodes from graph

`linestrings_from_graph()` — Convert graph to LINESTRING geometries
`distances_from_graph()` — Compute distance matrix from graph

OD Matrix Utilities

`melt_od_matrix()` — Convert origin-destination matrix to long format

Data

`africa_trade` — Average BACI HS96 2012-22 trade flows by section between 47 continental African countries
`africa_cities_ports` — The 453 largest (port-)cities in continental Africa within a 70km radius - from Krantz (2024), doi:10.1596/1813945010893
`africa_network` — African continental road network + extensions to optimally connect the 453 cities - from Krantz (2024), doi:10.1596/1813945010893
`africa_segments` — Primary segments derived from OpenStreetMap routes between the 453 cities - from Krantz (2024), doi:10.1596/1813945010893

Replication materials: `https://github.com/SebKrantz/OptimalAfricanRoads`

Details

The package uses efficient C implementations for critical path operations and leverages:

- `collapse` - Fast data transformations
- `geodist` - Fast geodesic distance computations
- `igraph` - Graph operations and shortest path algorithms
- `leaderCluster` - Fast spatial clustering for network simplification

Author(s)

Sebastian Krantz <`sebastian.krantz@graduateinstitute.ch`> and Kamol Roy <`kamol.roy08@gmail.com`>

References

Ben-Akiva, M., & Bierlaire, M. (1999). Discrete choice methods and their applications to short term travel decisions. In R. W. Hall (Ed.), *Handbook of Transportation Science* (pp. 5–33). Springer US. https://doi.org/10.1007/978-1-4615-5203-1_2

Cascetta, E. (2001). *Transportation systems engineering: Theory and methods*. Springer.

Ben-Akiva, M., & Lerman, S. R. (1985). *Discrete choice analysis: Theory and application to travel demand*. The MIT Press.

Ramming, M. S. (2002). *Network knowledge and route choice* (Doctoral dissertation). Massachusetts Institute of Technology.

Prato, C. G. (2009). Route choice modeling: Past, present and future research directions. *Journal of Choice Modelling, 2*(1), 65–100. https://doi.org/10.1016/S1755-5345(13)70005-8

AequilibiaE Python Documentation: https://www.aequilibrae.com/develop/python/route_choice/path_size_logit.html

`africa_cities_ports` *African Cities and International Ports*

Description

A spatial dataset containing 453 major African cities (population > 100,000) and international ports. Cities are deduplicated within 50-100km radii, with populations aggregated from nearby settlements. Port cities include cargo flow data from the World Bank Global Ports dataset.

Usage

```
data(africa_cities_ports)
```

Format

A Simple feature collection (sf object, also inheriting from data.table) with 453 POINT features and 12 fields:

city_country Character. Unique city-country identifier (e.g., "Cairo - Egypt", "Lagos - Nigeria").

city Character. City name.

country Character. Country name.

iso2 Character. ISO 3166-1 alpha-2 country code.

iso3 Character. ISO 3166-1 alpha-3 country code.

admin_name Character. Administrative region or province name.

capital Character. Capital status: "" (none), "admin" (administrative), "minor", or "primary" (national capital).

population Numeric. City population including nearby settlements within 30km.

port_locode Character. UN/LOCODE port identifier (empty string for non-port cities).

port_name Character. Official port name (empty string for non-port cities).

port_status Character. Port status code (empty string for non-port cities).

outflows Numeric. Outflows in TEU in Q1 of 2020 (NA for non-port cities). 51 cities have port outflow data.

geometry POINT. Spatial geometry in WGS 84 (EPSG:4326) coordinate reference system.

Details

The dataset was constructed by:

1. Selecting cities with population > 50,000 from Simplemaps World Cities database
2. Weighting by administrative importance (capital status)
3. Deduplicating within 50-100km radii, keeping largest weighted city
4. Aggregating populations from settlements within 30km
5. Matching with World Bank international ports within 30km

The bounding box spans from approximately 34S to 37N latitude and 17W to 49E longitude, covering continental Africa.

Source

City data from Simplemaps World Cities Database (`https://simplemaps.com/data/world-cities`). Port data from World Bank Global International Ports dataset (`https://datacatalog.worldbank.org/search/dataset/0038118`).

Dataset constructed for: Krantz, S. (2024). Optimal Investments in Africa's Road Network. Policy Research Working Paper 10893. World Bank. doi:10.1596/1813945010893. Replication materials: `https://github.com/SebKrantz/OptimalAfricanRoads`.

See Also

`africa_network`, `africa_trade`, flownet-package

Examples

```
library(sf)
data(africa_cities_ports)
head(africa_cities_ports)

# View largest cities
largest <- africa_cities_ports[order(-africa_cities_ports$population), ]
largest[1:10, c("city", "country", "population")]

# Filter port cities
ports <- africa_cities_ports[!is.na(africa_cities_ports$port_locode), ]
nrow(ports)  # 51 ports


plot(africa_cities_ports["population"])
```

`africa_network` *Trans-African Road Transport Network*

Description

A spatial dataset representing a discretized road transport network connecting major African cities and ports. The network combines existing road infrastructure (2,344 edges) with proposed new links (481 edges) identified through network efficiency analysis. Each edge contains distance, travel time, border crossing costs, terrain characteristics, and road upgrade cost estimates.

Usage

```
data(africa_network)
```

Format

A Simple feature collection (sf object) with 2,825 LINESTRING features and 28 fields:

from Integer. Origin node index (1 to 1,377).

to Integer. Destination node index (2 to 1,379).

from_ctry Character. Origin country ISO3 code (49 countries).

to_ctry Character. Destination country ISO3 code (49 countries).

FX Numeric. Origin node longitude.

FY Numeric. Origin node latitude.

TX Numeric. Destination node longitude.

TY Numeric. Destination node latitude.

sp_distance Numeric. Spherical (great-circle) distance in meters.

distance Numeric. Road distance in meters from OSRM routing.

duration Numeric. Travel duration in minutes from OSRM routing (NA for proposed links).

speed_kmh Numeric. Average speed in km/h (distance/duration) (NA for proposed links).

passes Numeric. Number of optimal inter-city routes passing through this edge (NA for proposed links).

gravity Numeric. Sum of population gravity weights from routes using this edge (NA for proposed links).

gravity_rd Numeric. Sum of road-distance-weighted gravity from routes (NA for proposed links).

border_dist Numeric. Additional distance for border crossing in meters (0 for domestic links).

total_dist Numeric. Total distance including border crossing penalty in meters.

border_time Numeric. Additional time for border crossing in minutes.

total_time Numeric. Total travel time including border crossing in minutes.

duration_100kmh Numeric. Hypothetical travel time at 100 km/h in minutes.

total_time_100kmh Numeric. Hypothetical total time at 100 km/h including border penalties.

rugg Numeric. Terrain ruggedness index along the edge.

pop_wpop Numeric. Population within corridor (WorldPop data).

pop_wpop_km2 Numeric. Population density per km2 along corridor.

cost_km Numeric. Estimated road construction/maintenance cost per km in USD.

upgrade_cat Character. Road upgrade category: "Nothing", "Asphalt Mix Resurfacing", "Mixed Works", "Upgrade", or NA.

ug_cost_km Numeric. Upgrade cost per km in USD.

add Logical. TRUE for proposed new links, FALSE for existing road network edges.

geometry LINESTRING. Spatial geometry in WGS 84 (EPSG:4326) coordinate reference system.

Details

The network was constructed through the following process:

1. Computing optimal routes between all city pairs within 2,000km using OSRM
2. Filtering routes using network efficiency criteria (alpha = 45 degrees, EU-grade efficiency)
3. Intersecting and aggregating overlapping route segments
4. Contracting the network to reduce complexity while preserving connectivity
5. Identifying proposed new links that would improve network route efficiency
6. Adding border crossing costs based on country pairs
7. Computing terrain, population, and road cost attributes

The `gravity` and `gravity_rd` fields measure edge importance based on the population gravity model: routes between larger, closer cities contribute more weight to edges they traverse.

The bounding box spans continental Africa from approximately 34S to 37N latitude and 17W to 49E longitude.

Source

Road network derived from OpenStreetMap via OSRM routing. Border crossing data from World Bank estimates. Terrain data from SRTM elevation models. Population data from WorldPop.

Dataset constructed for: Krantz, S. (2024). Optimal Investments in Africa's Road Network. Policy Research Working Paper 10893. World Bank. doi:10.1596/1813945010893. Replication materials: `https://github.com/SebKrantz/OptimalAfricanRoads`.

See Also

`africa_cities_ports`, `africa_segments`, `africa_trade`, flownet-package

Examples

```
library(sf)
data(africa_network)
head(africa_network)

# Existing vs proposed links
table(africa_network$add)

# Cross-border links
cross_border <- africa_network[africa_network$from_ctry != africa_network$to_ctry, ]
nrow(cross_border)

# Upgrade categories
table(africa_network$upgrade_cat, useNA = "ifany")


# Plot by gravity
plot(africa_network["gravity_rd"])
```

```
# Highlight proposed new links
plot(africa_network[africa_network$add, "geometry"], col = "red", add = TRUE)
```

`africa_segments` *Raw Network Segments for Trans-African Transport Network*

Description

A dataset containing 14,358 raw network segments representing intersected road routes between African cities. Each segment is defined by start and end coordinates with aggregate importance metrics. This dataset is provided to demonstrate how package functions like `consolidate_graph()` and `simplify_network()` can process messy segment data into clean analytical networks like `africa_network`.

Usage

```
data(africa_segments)
```

Format

A data frame with 14,358 rows and 7 columns:

FX Numeric. Start point longitude (range: -17.4 to 49.2).

FY Numeric. Start point latitude (range: -34.2 to 37.2).

TX Numeric. End point longitude (range: -17.0 to 49.1).

TY Numeric. End point latitude (range: -34.2 to 37.2).

passes Integer. Number of optimal inter-city routes passing through this segment. Range: 1 to 1,615, median: 46.

gravity Numeric. Sum of population gravity weights from routes using this segment. Computed as sum of (pop_origin * pop_destination / spherical_distance_km) / 1e9.

gravity_rd Numeric. Sum of road-distance-weighted gravity from routes. Computed as sum of (pop_origin * pop_destination / road_distance_m) / 1e9.

Details

This dataset represents an intermediate stage in network construction, after routes have been intersected but before network simplification. The segments have been simplified using `linestrings_from_graph()` to retain only start and end coordinates.

The segments can be used to demonstrate the flownet network processing workflow:

1. Convert segments to an sf LINESTRING object using `linestrings_from_graph()`
2. Apply `consolidate_graph()` to merge nearby nodes
3. Apply `simplify_network()` to remove intermediate nodes

The `passes` field indicates how many optimal city-to-city routes use each segment, serving as a measure of segment importance in the network. Higher values indicate segments that are critical for efficient inter-city connectivity.

Source

Derived from OpenStreetMap routing data via OSRM, processed through route intersection and aggregation.

Dataset constructed for: Krantz, S. (2024). Optimal Investments in Africa's Road Network. Policy Research Working Paper 10893. World Bank. doi:10.1596/1813945010893. Replication materials: `https://github.com/SebKrantz/OptimalAfricanRoads`.

See Also

`africa_network`, `consolidate_graph()`, `simplify_network()`, `linestrings_from_graph()`, flownet-package

Examples

```
data(africa_segments)
head(africa_segments)

# Summary statistics
summary(africa_segments[, c("passes", "gravity", "gravity_rd")])

# Segments with highest traffic
africa_segments[order(-africa_segments$passes), ][1:10, ]


# Convert to sf and plot
library(sf)
segments_sf <- linestrings_from_graph(africa_segments)
plot(segments_sf["passes"])
```

`africa_trade` *Intra-African Trade Flows by HS Section*

Description

A dataset containing bilateral trade flows between 47 African countries, aggregated by HS (Harmonized System) section. Values represent annual averages over 2012-2022 from the CEPII BACI database (HS96 nomenclature).

Usage

```
data(africa_trade)
```

Format

A data.table with 27,721 rows and 8 columns:

iso3_o Factor. Exporter (origin) country ISO 3166-1 alpha-3 code (47 countries).

iso3_d Factor. Importer (destination) country ISO 3166-1 alpha-3 code (47 countries).

section_code Integer. HS section code (1 to 21).

section_name Factor. HS section description (21 categories, e.g., "Live animals and animal products", "Mineral products", "Machinery and mechanical appliances...").

hs2_codes Factor. Comma-separated HS 2-digit codes within the section (e.g., "84, 85" for machinery).

value Numeric. Trade value in thousands of USD (current prices).

value_kd Numeric. Trade value in thousands of constant 2015 USD.

quantity Numeric. Trade quantity in metric tons.

Details

The dataset provides bilateral trade flows aggregated from HS 6-digit product codes (via HS 2-digit) to 21 HS sections. Trade values and quantities are annual averages computed over the 2012-2022 period.

HS sections cover broad product categories:

- Sections 1-5: Animal and vegetable products
- Sections 6-7: Chemical and plastic products
- Sections 8-14: Raw materials and manufactured goods
- Sections 15-16: Base metals and machinery
- Sections 17-21: Transport, instruments, and miscellaneous

Note: Some country pairs may have sparse trade relationships. Very small values indicate limited trade below typical reporting thresholds.

Source

CEPII BACI Database (HS96 nomenclature), Version 202401b, released 2024-04-09. Available at `https://www.cepii.fr/DATA_DOWNLOAD/baci/doc/baci_webpage.html`.

Reference: Gaulier, G. and Zignago, S. (2010). BACI: International Trade Database at the Product-Level. The 1994-2007 Version. CEPII Working Paper, N 2010-23.

See Also

`africa_cities_ports`, `africa_network`, flownet-package

Examples

```
data(africa_trade)
head(africa_trade)

# Number of trading pairs
length(unique(paste(africa_trade$iso3_o, africa_trade$iso3_d)))

# Total trade by section
aggregate(value ~ section_name, data = africa_trade, FUN = sum)

# Largest bilateral flows
africa_trade[order(-africa_trade$value), ][1:10, ]

# Trade between specific countries
subset(africa_trade, iso3_o == "ZAF" & iso3_d == "NGA")
```

`consolidate_graph` *Consolidate Graph*

Description

Consolidate a graph by removing intermediate nodes (nodes that occur exactly twice) and optionally dropping loop, duplicate, and singleton edges (leading to dead ends). This simplifies the network topology while preserving connectivity.

Usage

```
consolidate_graph(
  graph_df,
  directed = FALSE,
  drop.edges = c("loop", "duplicate", "single"),
  consolidate = TRUE,
  by = NULL,
  keep.nodes = NULL,
  ...,
  recursive = "full",
  verbose = TRUE
)
```

Arguments

`graph_df`	A data frame representing a graph with columns: `from` and `to` (node IDs), and optionally other columns to preserve. If coordinate columns (`FX`, `FY`, `TX`, `TY`) are present, they will be preserved and updated based on the consolidated node coordinates.
`directed`	Logical (default: FALSE). Whether the graph is directed.

`drop.edges`	Character vector (default: `c("loop", "duplicate", "single")`). Types of edges to drop: `"loop"` removes self-loops (edges where from == to), `"duplicate"` removes duplicate edges (same from-to pair), `"single"` removes edges leading to singleton nodes (nodes that occur only once). Set to `NULL` to keep all edges.
`consolidate`	Logical (default: TRUE). If TRUE, consolidates the graph by removing intermediate nodes (nodes that occur exactly twice) and merging connecting edges. If FALSE, only drops edges as specified in `drop.edges`.
`by`	Link characteristics to preserve/not consolidate across, passed as a one-sided formula or character vector of column names. Typically this includes attributes like *mode*, *type*, or *capacity* to ensure that only edges with the same characteristics are consolidated.
`keep.nodes`	Numeric vector (optional). Node IDs to preserve during consolidation, even if they occur exactly twice. Also used to preserve nodes when dropping singleton edges.
`...`	Arguments passed to `collap()` for aggregation across consolidated edges. The defaults are `FUN = fmean` for numeric columns and `catFUN = fmode` for categorical columns. Select columns using `cols` or use argument `custom = list(fmean = cols1, fsum = cols2, fmode = cols3)` to map different columns to specific aggregation functions. It is highly recommended to weight the aggregation (using `w = ~ weight_col`) by the length/cost of the edges.
`recursive`	One of `"none"/FALSE`, `"partial"` (recurse on dropping single edges and consolidation but only aggregate once), or `"full"/TRUE` (recursively consolidates and aggregates the graph until no further consolidation is possible). This ensures that long chains of intermediate nodes are fully consolidated in a single call.
`verbose`	Logical (default: TRUE). Whether to print messages about dropped edges and consolidation progress.

Details

This function simplifies a graph by:

- **Dropping edges**: Optionally removes self-loops, duplicate edges, and edges leading to singleton nodes (nodes that appear only once in the graph)
- **Consolidating nodes**: Removes intermediate nodes (nodes that occur exactly twice) by merging the two edges connected through them into a single longer edge
- **Aggregating attributes**: When edges are merged, attributes/columns are aggregated using `collap()`. The default aggregation is mean for numeric columns and mode for categorical columns.
- **Recursive consolidation**: If `recursive = TRUE`, the function continues consolidating until no more nodes can be consolidated, ensuring complete simplification

Consolidation is useful for simplifying network topology while preserving connectivity. For example, if node B connects A->B and B->C, it will be removed and replaced with A->C. With `recursive = TRUE`, long chains (A->B->C->D) are fully consolidated to A->D in a single call.

For undirected graphs, the algorithm also handles cases where a node appears twice as either origin or destination (circular cases).

If coordinate columns (`FX`, `FY`, `TX`, `TY`) are present in the input, they are preserved and updated based on the consolidated node coordinates from the original graph.

Value

A data frame representing the consolidated graph with:

- `edge` - Edge identifier (added as first column)
- All columns from `graph_df` (aggregated if consolidation occurred), excluding `from`, `to`, and optionally `FX`, `FY`, `TX`, `TY` (which are re-added if present in original)
- `from`, `to` - Node IDs (updated after consolidation)
- Coordinate columns (`FX`, `FY`, `TX`, `TY`) if present in original
- Attribute `"keep.edges"` - Indices of original edges that were kept
- Attribute `"gid"` - Edge group IDs mapping consolidated edges to original edges

See Also

create_undirected_graph simplify_network flownet-package

Examples

```
library(flownet)
library(sf)

# Convert segments to undirected graph
graph <- africa_segments |>
  linestrings_from_graph() |>
  linestrings_to_graph() |>
  create_undirected_graph(FUN = "fsum")

# Get nodes to preserve (city/port locations)
nodes <- nodes_from_graph(graph, sf = TRUE)
nearest_nodes <- nodes$node[st_nearest_feature(africa_cities_ports, nodes)]

# Consolidate graph, preserving city nodes
graph_cons <- consolidate_graph(graph, keep = nearest_nodes, w = ~ passes)

# Consolidated graph has fewer edges
c(original = nrow(graph), consolidated = nrow(graph_cons))
```

`create_undirected_graph` *Create Undirected Graph*

Description

Convert a directed graph to an undirected graph by normalizing edges and aggregating duplicate connections.

Usage

```
create_undirected_graph(graph_df, by = NULL, ...)
```

Arguments

`graph_df`	A data frame representing a directed graph including columns: `from`, `to`, and (optionally) `edge`, `FX`, `FY`, `TX`, `TY`.
`by`	Link characteristics to preserve/not aggregate across, passed as a one-sided formula or character vector of column names. Typically this includes attributes like *mode*, *type*, or *capacity* to ensure that only edges with the same characteristics are aggregated.
`...`	Arguments passed to `collap()` for aggregation across duplicated (diretional) edges. The defaults are `FUN = fmean` for numeric columns and `catFUN = fmode` for categorical columns. Select columns using `cols` or use argument `custom = list(fmean = cols1, fsum = cols2, fmode = cols3)` to map different columns to specific aggregation functions. You can weight the aggregation (using `w = ~ weight_col`).

Details

This function converts a directed graph to an undirected graph by:

- Normalizing edge directions so that `from` < `to` for all edges
- Collapsing duplicate edges (same `from` and `to` nodes)
- For spatial/identifier columns (`edge`, `FX`, `FY`, `TX`, `TY`), taking the first value from duplicates
- For aggregation columns, `collap()` will be applied.

Value

A data frame representing an undirected graph with:

- `edge` - Edge identifier (first value from duplicates)
- `from` - Starting node ID (normalized to be < `to`)
- `to` - Ending node ID (normalized to be > `from`)
- `FX` - Starting node X-coordinate (first value from duplicates)
- `FY` - Starting node Y-coordinate (first value from duplicates)
- `TX` - Ending node X-coordinate (first value from duplicates)
- `TY` - Ending node Y-coordinate (first value from duplicates)
- Aggregated columns

Examples

```
library(flownet)

# Convert segments to graph and make undirected
graph <- africa_segments |>
  linestrings_from_graph() |>
  linestrings_to_graph()
graph_undir <- create_undirected_graph(graph, FUN = "fsum")

# Fewer edges after removing directional duplicates
c(directed = nrow(graph), undirected = nrow(graph_undir))
```

`distances_from_graph` *Compute Distance Matrix from Graph*

Description

Compute a distance matrix for all node pairs in a graph using cppRouting.

Usage

```
distances_from_graph(graph_df, directed = FALSE, cost.column = "cost", ...)
```

Arguments

`graph_df`	A data frame representing a graph with columns: `from`, `to`, and `cost`.
`directed`	Logical (default: FALSE). If TRUE, treats the graph as directed; if FALSE, treats it as undirected.
`cost.column`	Character string (optional). Name of the cost column in `graph_df`. Alternatively, a numeric vector of edge costs with length equal to `nrow(graph_df)`.
`...`	Additional arguments passed to `distances()`, such as `v` (from) and `to` to compute paths between specific nodes.

Details

This function:

- Converts the graph data frame to a cppRouting graph object
- Contracts the graph for efficient distance computation
- Computes the distance matrix for all node pairs using the specified algorithm

Value

A matrix of distances between all node pairs, where rows and columns correspond to node IDs. The matrix contains the shortest path distances (based on the `cost` column) between all pairs of nodes.

Examples

```
library(flownet)

# Create a simple graph
graph <- data.frame(
  from = c(1, 2, 2, 3),
  to = c(2, 3, 4, 4),
  cost = c(1, 2, 3, 1)
)

# Compute distance matrix
dmat <- distances_from_graph(graph, cost.column = "cost")
dmat
```

`linestrings_from_graph`   *Convert Graph to Linestrings*

Description

Convert a graph data frame with node coordinates to an sf object with LINESTRING geometries.

Usage

```
linestrings_from_graph(graph_df, crs = 4326)
```

Arguments

`graph_df`	A data frame representing a graph with columns: `FX`, `FY`, `TX`, `TY` (starting and ending node coordinates), and optionally other columns to preserve.
`crs`	Numeric or character (default: 4326). Coordinate reference system to assign to the output sf object.

Details

This function is the inverse operation of `linestrings_to_graph`. It:

- Creates LINESTRING geometries from node coordinates (`FX`, `FY`, `TX`, `TY`)
- Removes the coordinate columns from the output
- Preserves all other columns from the input graph data frame
- Returns an sf object suitable for spatial operations and visualization

Value

An sf data frame with LINESTRING geometry, containing all columns from `graph_df` except `FX`, `FY`, `TX`, and `TY`. Each row represents an edge as a LINESTRING connecting the from-node (`FX`, `FY`) to the to-node (`TX`, `TY`).

See Also

linestrings_to_graph flownet-package

Examples

```
library(flownet)
library(sf)

# Convert segments data frame to sf LINESTRING object
segments_sf <- linestrings_from_graph(africa_segments)
class(segments_sf)
head(segments_sf)


# Plot segments colored by route importance
plot(segments_sf["passes"])
```

`linestrings_to_graph` *Convert Linestring to Graph*

Description

Convert Linestring to Graph

Usage

```
linestrings_to_graph(
  lines,
  digits = 6,
  keep.cols = is.atomic,
  compute.length = TRUE
)
```

Arguments

`lines`	An sf data frame of LINESTRING geometries.
`digits`	Numeric rounding applied to coordinates (to ensure that matching points across different linestrings is not impaired by numeric precision issues). Set to `NA`/`Inf`/`FALSE` to disable.
`keep.cols`	Character vector of column names to keep from the input data frame.
`compute.length`	Applies `st_length()` to and saves it as an additional column named `".length"`.

Value

A data.frame representing the graph with columns:

- `edge` - Edge identifier
- `from` - Starting node ID
- `FX` - Starting node X-coordinate (longitude)
- `FY` - Starting node Y-coordinate (latitude)
- `to` - Ending node ID
- `TX` - Ending node X-coordinate (longitude)
- `TY` - Ending node Y-coordinate (latitude)

See Also

simplify_network flownet-package

Examples

```
library(flownet)
library(sf)

# Load existing network edges (exclude proposed new links)
africa_net <- africa_network[!africa_network$add, ]

# Convert network LINESTRING geometries to graph
graph <- linestrings_to_graph(africa_net)
head(graph)

# Graph contains edge, from/to nodes, and coordinates
names(graph)
```

`melt_od_matrix` *Melt Origin-Destination Matrix to Long Format*

Description

Convert an origin-destination (OD) matrix to a long-format data frame with columns `from`, `to`, and `flow`.

Usage

```
melt_od_matrix(od_matrix, nodes = NULL, sort = TRUE)
```

Arguments

`od_matrix`	A numeric matrix with origin-destination flows. Rows represent origins, columns represent destinations. The matrix should be square (same number of rows and columns).
`nodes`	(Optional) Numeric or integer vector of node IDs in the graph matching the rows and columns of the matrix. If provided, must have length equal to both `nrow(od_matrix)` and `ncol(od_matrix)`. When `nodes` is provided, these IDs are used directly, ignoring row and column names. This is particularly useful when mapping zone-based OD matrices to graph node IDs (e.g., using `st_nearest_feature` to find nearest graph nodes). If omitted, row and column names (if present) will be used as node IDs, coerced to integer if possible. If names are not available or cannot be coerced to integer, sequential integers will be used.
`sort`	Sort long OD-matrix in ascending order of from and to columns. This can have computational benefits, e.g., when multithreading with `method = "AoN"`.

Details

This function converts a square OD matrix to long format, which is required by `run_assignment()`. The behavior depends on whether `nodes` is provided:

When `nodes` is provided:

- The `nodes` vector is used directly as node IDs for both origins and destinations
- Row and column names are ignored (but must match if both are present)
- This is the recommended approach when working with zone-based OD matrices that need to be mapped to graph nodes, as it ensures the node IDs match those in the graph

When `nodes` is omitted:

- Row and column names are extracted from the matrix (if available)
- Names are coerced to integer if possible; if coercion fails or names are missing, sequential integers are used
- This approach works well when the matrix row/column names already correspond to graph node IDs

In both cases, the function:

- Creates a long-format data frame with all origin-destination pairs
- Filters out non-finite and zero flow values

The function is useful for converting OD matrices to the long format required by `run_assignment()`.

Value

A data frame with columns:

- `from` - Origin node ID (integer)
- `to` - Destination node ID (integer)
- `flow` - Flow value (numeric)

Only rows with finite, positive flow values are included.

See Also

africa_cities_ports, africa_network, nodes_from_graph(), run_assignment(), flownet-package

Examples

```
library(flownet)
library(sf)

# Load existing network and convert to graph
africa_net <- africa_network[!africa_network$add, ]
graph <- linestrings_to_graph(africa_net)
nodes <- nodes_from_graph(graph, sf = TRUE)

# Map cities/ports to nearest network nodes
nearest_nodes <- nodes$node[st_nearest_feature(africa_cities_ports, nodes)]

# Example 1: Simple gravity-based OD matrix
od_mat <- outer(africa_cities_ports$population, africa_cities_ports$population) / 1e12
dimnames(od_mat) <- list(nearest_nodes, nearest_nodes)
od_long <- melt_od_matrix(od_mat)
head(od_long)

# Example 2: Using nodes argument (when matrix has zone IDs, not node IDs)
# Here zones are 1:n_cities, nodes argument maps them to graph nodes
dimnames(od_mat) <- NULL
od_long2 <- melt_od_matrix(od_mat, nodes = nearest_nodes)
head(od_long2)
```

---

nodes_from_graph *Extract Nodes from Graph*

Description

Extract unique nodes with their coordinates from a graph data frame.

Usage

```
nodes_from_graph(graph_df, sf = FALSE, crs = 4326)
```

Arguments

graph_df	A data frame representing a graph with columns: from, to, FX, FY, TX, TY.
sf	Logical. If TRUE, returns result as an sf POINT object. Default: FALSE.
crs	Coordinate reference system for sf output; default is 4326.

Details

This function extracts all unique nodes from both the `from` and `to` columns of the graph, along with their corresponding coordinates. Duplicate nodes are removed, keeping only unique node IDs with their coordinates.

Value

A data frame (or sf object if `sf = TRUE`) with unique nodes and coordinates:

- `node` - Node ID
- `X` - Node X-coordinate (typically longitude)
- `Y` - Node Y-coordinate (typically latitude)

Result is sorted by node ID.

Examples

```
library(flownet)
library(sf)

# Load existing network edges and convert to graph
africa_net <- africa_network[!africa_network$add, ]
graph <- linestrings_to_graph(africa_net)

# Extract nodes from graph
nodes <- nodes_from_graph(graph)
head(nodes)

# Get nodes as sf POINT object for spatial operations
nodes_sf <- nodes_from_graph(graph, sf = TRUE)
class(nodes_sf)

# Find nearest network nodes to cities/ports
nearest_nodes <- nodes_sf$node[st_nearest_feature(africa_cities_ports, nodes_sf)]
head(nearest_nodes)
```

`normalize_graph` *Normalize Graph Node IDs*

Description

Normalize node IDs in a graph to be consecutive integers starting from 1. This is useful for ensuring compatibility with graph algorithms that require sequential node IDs.

Usage

```
normalize_graph(graph_df)
```

Arguments

`graph_df`	A data frame representing a graph with columns: `from` and `to` (node IDs).

Details

This function:

- Extracts all unique node IDs from both `from` and `to` columns
- Sorts them in ascending order
- Remaps the original node IDs to sequential integers (1, 2, 3, ...)
- Updates both `from` and `to` columns with the normalized IDs

Normalization is useful when:

- Node IDs are non-consecutive (e.g., 1, 5, 10, 20)
- Node IDs are non-numeric or contain gaps
- Graph algorithms require sequential integer node IDs starting from 1

Note: This function only normalizes the node IDs; it does not modify the graph structure or any other attributes. The mapping preserves the relative ordering of nodes.

Value

A data frame with the same structure as `graph_df`, but with `from` and `to` columns remapped to consecutive integer IDs starting from 1. All other columns are preserved unchanged.

See Also

nodes_from_graph flownet-package

Examples

```
library(flownet)

# Create graph with non-consecutive node IDs
graph <- data.frame(
  from = c(10, 20, 20),
  to = c(20, 30, 40),
  cost = c(1, 2, 3)
)

# Normalize to consecutive integers (1, 2, 3, 4)
graph_norm <- normalize_graph(graph)
graph_norm
```

`run_assignment` *Run Traffic Assignment*

Description

Assign traffic flows to network edges using either Path-Sized Logit (PSL) or All-or-Nothing (AoN) assignment methods.

Usage

```
run_assignment(
  graph_df,
  od_matrix_long,
  directed = FALSE,
  cost.column = "cost",
  method = c("PSL", "AoN"),
  beta = 1,
  ...,
  detour.max = 1.5,
  angle.max = 90,
  unique.cost = TRUE,
  npaths.max = Inf,
  dmat.max.size = 10000^2,
  return.extra = NULL,
  verbose = TRUE,
  nthreads = 1L
)

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

Arguments

`graph_df`	A data.frame with columns `from`, `to`, and optionally a cost column. Represents the network graph with edges between nodes.
`od_matrix_long`	A data.frame with columns `from`, `to`, and `flow`. Represents the origin-destination matrix in long format with flow values.
`directed`	Logical (default: FALSE). Whether the graph is directed.
`cost.column`	Character string (default: "cost") or numeric vector. Name of the cost column in `graph_df`, or a numeric vector of edge costs with length equal to `nrow(graph_df)`. The cost values are used to compute shortest paths and determine route probabilities.
`method`	Character string (default: "PSL"). Assignment method: • `"PSL"`: Path-Sized Logit model considering multiple routes with overlap correction

- `"AoN"`: All-or-Nothing assignment, assigns all flow to the shortest path (faster but no route alternatives)

`beta`	Numeric (default: 1). Path-sized logit parameter (beta_PSL). Only used for PSL method.
`...`	Additional arguments (currently ignored).
`detour.max`	Numeric (default: 1.5). Maximum detour factor for alternative routes (applied to shortest paths cost). Only used for PSL method. This is a key parameter controlling the execution time of the algorithm: considering more routes (higher `detour.max`) substantially increases computation time.
`angle.max`	Numeric (default: 90). Maximum detour angle (in degrees, two sided). Only used for PSL method. I.e., nodes not within this angle measured against a straight line from origin to destination node will not be considered for detours.
`unique.cost`	Logical (default: TRUE). Deduplicates paths based on the total cost prior to generating them. Only used for PSL method. Since multiple 'intermediate nodes' may be on the same path, there is likely a significant number of duplicate paths which this option removes.
`npaths.max`	Integer (default: Inf). Maximum number of paths to compute per OD-pair. Only used for PSL method. If the number of paths exceeds this number, a random sample will be taken from all but the shortest path.
`dmat.max.size`	Integer (default: `1e4^2`). Maximum size of distance matrices (both shortest paths and geodesic) to precompute. If smaller than `n_nodes^2`, then the full matrix is precomputed. Otherwise, it is computed in chunks as needed, where each chunk has `dmat.max.size` elements. Only used for PSL method.
`return.extra`	Character vector specifying additional results to return. Use `"all"` to return all available extras for the selected method.

Option	**PSL**	**AoN**	**Description**
`"graph"`	Yes	Yes	The igraph graph object
`"paths"`	Yes	Yes	PSL: list of lists of edge indices (multiple routes per OD); AoN: list of edge index vectors (one p
`"edges"`	Yes	No	List of edge indices used for each OD pair
`"counts"`	Yes	Yes	PSL: list of edge visit counts per OD; AoN: integer vector of global edge traversal counts
`"costs"`	Yes	Yes	PSL: list of path costs per OD; AoN: numeric vector of shortest path costs
`"weights"`	Yes	No	List of path weights (probabilities) for each OD pair

`verbose`	Logical (default: TRUE). Show progress bar and intermediate steps completion status?
`nthreads`	Integer (default: 1L). Number of threads (daemons) to use for parallel processing with `mirai`. Should not exceed the number of logical processors.
`x`	An object of class `flownet`, typically returned by `run_assignment`.

Details

This function performs traffic assignment using one of two methods: **All-or-Nothing (AoN)** is fast but assigns all flow to a single shortest path; **Path-Sized Logit (PSL)** considers multiple routes with overlap correction for more realistic flow distribution.

All-or-Nothing (AoN) Method: A simple assignment method that assigns all flow from each OD pair to the single shortest path. This is much faster than PSL but does not consider route alternatives or overlaps. Parameters `detour.max`, `angle.max`, `unique.cost`, `npaths.max`, `beta`, and `dmat.max.size` are ignored for AoN.

Path-Sized Logit (PSL) Method: A more sophisticated assignment method that considers multiple alternative routes and accounts for route overlap when assigning flows. The PSL model adjusts choice probabilities based on how much each route overlaps with other alternatives, preventing overestimation of flow on shared segments. The `beta` parameter controls the sensitivity to overlap.

PSL Model Formulation: The probability P_k of choosing route k from the set of alternatives K is:

$$P_k = \frac{e^{V_k}}{\sum_{j \in K} e^{V_j}}$$

where the utility V_k is defined as:

$$V_k = -C_k + \beta_{PSL} \ln(PS_k)$$

Here C_k is the generalized cost of route k, β_{PSL} is the path-size parameter (the `beta` argument), and PS_k is the path-size factor.

The path-size factor quantifies route uniqueness:

$$PS_k = \frac{1}{C_k} \sum_{a \in \Gamma_k} \frac{c_a}{\delta_a}$$

where Γ_k is the set of edges in path k, c_a is the cost of edge a, and δ_a is the number of alternative routes using edge a.

If a path is unique ($\delta_a = 1$ for all edges), then $PS_k = 1$ and the model reduces to standard MNL. For overlapping routes, $PS_k < 1$ and $\ln(PS_k) < 0$, so a positive `beta` penalizes overlap. Higher `beta` values strengthen penalization; `beta = 0` gives standard MNL behavior.

For more information about the PSL model consult some of the references below.

Route Enumeration Algorithm: For each origin-destination pair, the algorithm identifies alternative routes as follows:

1. Compute the shortest path cost from origin to destination.
2. For each potential intermediate node, calculate the total cost of going origin -> intermediate -> destination.
3. Keep only routes where total cost is within `detour.max` times the shortest path cost.
4. If `angle.max` is specified, filter to intermediate nodes that lie roughly in the direction of the destination (within the specified angle).
5. If `unique.cost = TRUE`, remove duplicate routes based on total cost.
6. Compute the actual paths and filter out those with duplicate edges (where the intermediate node is approached and departed via the same edge).

This pre-selection using distance matrices speeds up route enumeration considerably by avoiding computation of implausible paths.

Coordinate-Based Filtering: When `angle.max` is specified and `graph_df` contains coordinate columns (`FX`, `FY`, `TX`, `TY`), the function uses geographic distance calculations to restrict detours. Only intermediate nodes that are (a) closer to the origin than the destination is, and (b) within the specified angle from the origin-destination line are considered. This improves both computational efficiency and route realism by excluding geographically implausible detours.

Value

A list of class `"flownet"` containing:

- `call` - The function call
- `final_flows` - Numeric vector of assigned flows for each edge (same length as `nrow(graph_df)`)
- `od_pairs_used` - Indices of OD pairs with valid flows
- Additional elements as specified in `return.extra`:
  - `graph` - The igraph graph object
  - `paths` - For PSL: list of lists of edge indices (multiple routes per OD pair); for AoN: list of edge index vectors (one shortest path per OD pair)
  - `edges` - List of edge indices used for each OD pair (PSL only)
  - `edge_counts` - For PSL: list of edge visit counts per OD pair; for AoN: integer vector of global edge traversal counts
  - `path_costs` - For PSL: list of path costs per OD pair; for AoN: numeric vector of shortest path costs
  - `path_weights` - List of path weights (probabilities) for each OD pair (PSL only)

References

Ben-Akiva, M., & Bierlaire, M. (1999). Discrete choice methods and their applications to short term travel decisions. In R. W. Hall (Ed.), *Handbook of Transportation Science* (pp. 5–33). Springer US. https://doi.org/10.1007/978-1-4615-5203-1_2

Cascetta, E. (2001). *Transportation systems engineering: Theory and methods*. Springer.

Ben-Akiva, M., & Lerman, S. R. (1985). *Discrete choice analysis: Theory and application to travel demand*. The MIT Press.

Ramming, M. S. (2002). *Network knowledge and route choice* (Doctoral dissertation). Massachusetts Institute of Technology.

Prato, C. G. (2009). Route choice modeling: Past, present and future research directions. *Journal of Choice Modelling, 2*(1), 65–100. https://doi.org/10.1016/S1755-5345(13)70005-8

AequilibiaE Python Documentation: https://www.aequilibrae.com/develop/python/route_choice/path_size_logit.html

See Also

flownet-package

Examples

```
library(flownet)
library(collapse)
library(sf)

# Load existing network edges (exclude proposed new links)
africa_net <- africa_network[!africa_network$add, ]

# Convert to graph (use atomic_elem to drop sf geometry, qDF for data.frame)
graph <- atomic_elem(africa_net) |> qDF()
nodes <- nodes_from_graph(graph, sf = TRUE)

# Map cities/ports to nearest network nodes
nearest_nodes <- nodes$node[st_nearest_feature(africa_cities_ports, nodes)]

# Simple gravity-based OD matrix
od_mat <- outer(africa_cities_ports$population, africa_cities_ports$population) / 1e12
dimnames(od_mat) <- list(nearest_nodes, nearest_nodes)
od_matrix_long <- melt_od_matrix(od_mat)

# Run Traffic Assignment (All-or-Nothing method)
result_aon <- run_assignment(graph, od_matrix_long, cost.column = "duration",
                             method = "AoN", return.extra = "all")
print(result_aon)

# Run Traffic Assignment (Path-Sized Logit method)
# Note: PSL is slower but produces more realistic flow distribution
result_psl <- run_assignment(graph, od_matrix_long, cost.column = "duration",
                             method = "PSL", nthreads = 1L,
                             return.extra = c("edges", "counts", "costs", "weights"))
print(result_psl)

# Visualize AoN Results
africa_net$final_flows_log10 <- log10(result_psl$final_flows + 1)
plot(africa_net["final_flows_log10"], main = "PSL Assignment")



# --- Trade Flow Disaggregation Example ---
# Disaggregate country-level trade to city-level using population shares

# Compute each city's share of its country's population
city_pop <- africa_cities_ports |> atomic_elem() |> qDF() |>
  fcompute(node = nearest_nodes,
           city = qF(city_country),
           pop_share = fsum(population, iso3, TRA = "/"),
           keep = "iso3")

# Aggregate trade to country-country level and disaggregate to cities
trade_agg <- africa_trade |> collap(quantity ~ iso3_o + iso3_d, fsum)
od_matrix_trade <- trade_agg |>
  join(city_pop |> add_stub("_o", FALSE), multiple = TRUE) |>
```

```
    join(city_pop |> add_stub("_d", FALSE), multiple = TRUE) |>
    fmutate(flow = quantity * pop_share_o * pop_share_d) |>
    frename(from = node_o, to = node_d) |>
    fsubset(flow > 0 & from != to)

  # Run AoN assignment with trade flows
  result_trade_aon <- run_assignment(graph, od_matrix_trade, cost.column = "duration",
                                     method = "AoN", return.extra = "all")
  print(result_trade_aon)

  # Visualize trade flow results
  africa_net$trade_flows_log10 <- log10(result_trade_aon$final_flows + 1)
  plot(africa_net["trade_flows_log10"], main = "Trade Flow Assignment (AoN)")

  # Run PSL assignment with trade flows (nthreads can be increased for speed)
  result_trade_psl <- run_assignment(graph, od_matrix_trade, cost.column = "duration",
                                     method = "PSL", nthreads = 1L,
                                 return.extra = c("edges", "counts", "costs", "weights"))
  print(result_trade_psl)

  # Compare PSL vs AoN: PSL typically shows more distributed flows
  africa_net$trade_flows_psl_log10 <- log10(result_trade_psl$final_flows + 1)
  plot(africa_net["trade_flows_psl_log10"], main = "Trade Flow Assignment (PSL)")
```

simplify_network *Simplify Network*

Description

Simplify a network graph using shortest paths or node clustering methods.

Usage

```
simplify_network(
  graph_df,
  nodes,
  method = c("shortest-paths", "cluster"),
  directed = FALSE,
  cost.column = "cost",
  by = NULL,
  radius_km = list(nodes = 7, cluster = 20),
  ...
)
```

Arguments

`graph_df`	A data.frame with columns `from` and `to` representing the graph edges. For the cluster method, the graph must also have columns `FX`, `FY`, `TX`, `TY` representing node coordinates.
`nodes`	For `method = "shortest-paths"`: either an atomic vector of node IDs, or a data.frame with columns `from` and `to` specifying origin-destination pairs. For `method = "cluster"`: an atomic vector of node IDs to preserve. These nodes will be kept as cluster centroids, and nearby nodes (within `radius_km$nodes`) will be assigned to their clusters. Remaining nodes are clustered using `leaderCluster`.
`method`	Character string (default: "shortest-paths"). Method to use for simplification: `"shortest-paths"` computes shortest paths between nodes and keeps only traversed edges; `"cluster"` clusters nodes using the `leaderCluster` algorithm and contracts the graph.
`directed`	Logical (default: FALSE). Whether the graph is directed. For `method = "shortest-paths"`: controls path computation direction. For `method = "cluster"`: if TRUE, A->B and B->A remain as separate edges after contraction; if FALSE, edges are normalized so that `from` < `to` before grouping.
`cost.column`	Character string (default: "cost"). Name of the cost column in `graph_df`. Alternatively, a numeric vector of edge costs with length equal to `nrow(graph_df)`. With `method = "cluster"`, a numeric vector of node weights matching `nodes_from_graph(graph_df)` can be provided.
`by`	Link characteristics to preserve/not simplify across, passed as a one-sided formula or character vector of column names. Typically includes attributes like *mode*, *type*, or *capacity*. For `method = "shortest-paths"`: paths are computed separately for each group defined by `by`, with edges not in the current group penalized (cost multiplied by 100) to compel mode-specific routes. For `method = "cluster"`: edges are grouped by `from`, `to`, AND `by` columns, preventing consolidation across different modes/types.
`radius_km`	Named list with elements `nodes` (default: 7) and `cluster` (default: 20). Only used for `method = "cluster"`. `nodes`: radius in kilometers around preserved nodes. Graph nodes within this radius will be assigned to the nearest preserved node's cluster. `cluster`: radius in kilometers for clustering remaining nodes using leaderCluster.
`...`	For `method = "cluster"`: additional arguments passed to `collap` for edge attribute aggregation.

Details

Method: "shortest-paths"

- Validates that all origin and destination nodes exist in the network
- Computes shortest paths from each origin to all destinations using igraph
- Marks all edges that are traversed by at least one shortest path
- Returns only the subset of edges that were traversed
- If `nodes` is a data frame with `from` and `to` columns, paths are computed from each unique origin to its specified destinations

Method: "cluster"

- Requires the graph to have spatial coordinates (`FX`, `FY`, `TX`, `TY`)
- If `nodes` is provided, these nodes are preserved as cluster centroids
- Nearby nodes (within `radius_km$nodes` km) are assigned to the nearest preserved node
- Remaining nodes are clustered using `leaderCluster` with `radius_km$cluster` as the clustering radius
- For each cluster, the node closest to the cluster centroid is selected as representative
- The graph is contracted by mapping all nodes to their cluster representatives
- Self-loops (edges where both endpoints map to the same cluster) are dropped
- For undirected graphs (`directed = FALSE`), edges are normalized so `from` < `to`, merging opposite-direction edges; for directed graphs, A->B and B->A remain separate
- Edge attributes are aggregated using `collap` (default: mean for numeric, mode for categorical); customize via `...`

Value

A data.frame containing the simplified graph with:

- For `method = "shortest-paths"`:
  - All columns from the input `graph_df` (for edges that were kept)
  - Attribute `"edges"`: integer vector of edge indices from the original graph
  - Attribute `"edge_counts"`: integer vector indicating how many times each edge was traversed
- For `method = "cluster"`:
  - edge - New edge identifier
  - `from`, `to` - Cluster centroid node IDs
  - `FX`, `FY`, `TX`, `TY` - Coordinates of cluster centroid nodes
  - Aggregated edge attributes from the original graph
  - Attribute `"group.id"`: mapping from original edges to simplified edges
  - Attribute `"group.starts"`: start indices of each group
  - Attribute `"group.sizes"`: number of original edges per simplified edge

Examples

```
library(flownet)
library(sf)

# Convert segments to undirected graph
graph <- africa_segments |>
  linestrings_from_graph() |>
  linestrings_to_graph() |>
  create_undirected_graph(FUN = "fsum")

# Get city/port nodes to preserve
nodes_df <- nodes_from_graph(graph, sf = TRUE)
```

```
nearest_nodes <- nodes_df$node[st_nearest_feature(africa_cities_ports, nodes_df)]

# Initial consolidation
graph <- consolidate_graph(graph, keep = nearest_nodes, w = ~ passes)

# Method 1: Shortest-paths simplification (keeps only traversed edges)
graph_simple <- simplify_network(graph, nearest_nodes,
                                 method = "shortest-paths",
                                 cost.column = ".length")
nrow(graph_simple)  # Reduced number of edges


# Method 2: Cluster-based simplification (contracts graph spatially)
# Compute node weights for clustering
node_weights <- collapse::rowbind(
  collapse::fselect(graph, node = from, gravity_rd),
  collapse::fselect(graph, to, gravity_rd),
  use.names = FALSE) |>
  collapse::collap(~ node, "fsum")

graph_cluster <- simplify_network(graph, nearest_nodes,
                                  method = "cluster",
                                  cost.column = node_weights$gravity_rd,
                                  radius_km = list(nodes = 30, cluster = 27),
                                  w = ~ passes)
nrow(graph_cluster)
```

Index