Package 'spopt'

May 9, 2026

Title Spatial Optimization for Regionalization, Facility Location, and Market Analysis

Version 0.1.2

Date 2026-04-16

Description Implements spatial optimization algorithms across several problem families including contiguity-constrained regionalization, discrete facility location, market share analysis, and least-cost corridor and route optimization over raster cost surfaces. Facility location problems also accept user-supplied network travel-time matrices. Uses a 'Rust' backend via 'extendr' for graph and routing algorithms, and the 'HiGHS' solver via the 'highs' package for facility location mixed-integer programs. Method-level references are provided in the documentation of the individual functions.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

Imports sf, spdep, Matrix, highs

Suggests testthat (>= 3.0.0), terra, dodgr, r5r, knitr, rmarkdown, quarto, tidycensus, tidyverse, mapgl

Config/testthat/edition 3

Config/rextendr/version 0.4.2.9000

SystemRequirements Cargo (Rust's package manager), rustc >= 1.70

URL https://walker-data.com/spopt-r/, https://github.com/walkerke/spopt-r

BugReports https://github.com/walkerke/spopt-r/issues

Depends R (>= 4.2)

LazyData true

NeedsCompilation yes

Author Kyle Walker [aut, cre], PySAL Developers [cph] (Original Python spopt library)

Maintainer Kyle Walker <kyle@walker-data.com>

Repository CRAN

Date/Publication 2026-04-22 07:40:02 UTC

Contents

azp *Automatic Zoning Procedure (AZP)*

Description

Performs regionalization using the Automatic Zoning Procedure algorithm. AZP uses local search to minimize within-region heterogeneity while maintaining spatial contiguity. Three variants are available: basic (greedy), tabu search, and simulated annealing.

Usage

```
azp(
  data,
  attrs = NULL,
  n_regions,
  weights = "queen",
  bridge_islands = FALSE,
  method = c("tabu", "basic", "sa"),
  max_iterations = 100L,
  tabu_length = 10L,
  cooling_rate = 0.99,
  initial_temperature = 0,
  scale = TRUE,
  seed = NULL,
  verbose = FALSE
)
```

Arguments

`data`	An sf object with polygon or point geometries.
`attrs`	Character vector of column names to use for clustering (e.g., `c("var1", "var2")`). If NULL, uses all numeric columns.
`n_regions`	Integer. Number of regions (clusters) to create.
`weights`	Spatial weights specification. Can be: • `"queen"` (default): Polygons sharing any boundary point are neighbors • `"rook"`: Polygons sharing an edge are neighbors • An `nb` object from spdep or created with `sp_weights()` • A list for other weight types: `list(type = "knn", k = 6)` for k-nearest neighbors, or `list(type = "distance", d = 5000)` for distance-based weights
`bridge_islands`	Logical. If TRUE, automatically connect disconnected components (e.g., islands) using nearest-neighbor edges. If FALSE (default), the function will error when the spatial weights graph is disconnected.
`method`	Character. Optimization method: "basic" (greedy local search), "tabu" (tabu search), or "sa" (simulated annealing). Default is "tabu".

`max_iterations`	Integer. Maximum number of iterations (default 100).
`tabu_length`	Integer. Length of tabu list for tabu method (default 10).
`cooling_rate`	Numeric. Cooling rate for SA method, between 0 and 1 (default 0.99).
`initial_temperature`	Numeric. Initial temperature for SA method. If 0 (default), automatically set based on initial objective.
`scale`	Logical. If TRUE (default), standardize attributes before clustering.
`seed`	Optional integer for reproducibility.
`verbose`	Logical. Print progress messages.

Details

The Automatic Zoning Procedure (AZP) was introduced by Openshaw (1977) and refined by Openshaw & Rao (1995). It is a local search algorithm that:

1. Starts with an initial random partition into n_regions
2. Iteratively moves border areas between regions to reduce heterogeneity
3. Maintains spatial contiguity throughout
4. Terminates when no improving moves are found

Three variants are available:

- **basic**: Greedy local search that only accepts improving moves
- **tabu**: Tabu search that can accept non-improving moves to escape local optima, with a tabu list preventing cycling
- **sa**: Simulated annealing that accepts worse moves with decreasing probability as temperature cools

Value

An sf object with a `.region` column containing cluster assignments. Metadata is stored in the "spopt" attribute, including:

- algorithm: "azp"
- method: The optimization method used
- n_regions: Number of regions created
- objective: Total within-region sum of squared deviations
- solve_time: Time to solve in seconds

References

Openshaw, S. (1977). A geographical solution to scale and aggregation problems in region-building, partitioning and spatial modelling. Transactions of the Institute of British Geographers, 2(4), 459-472.

Openshaw, S., & Rao, L. (1995). Algorithms for reengineering 1991 Census geography. Environment and Planning A, 27(3), 425-446.

Examples

```
library(sf)
nc <- st_read(system.file("shape/nc.shp", package = "sf"))

# Basic AZP with 8 regions
result <- azp(nc, attrs = c("SID74", "SID79"), n_regions = 8)

# Tabu search (often finds better solutions)
result <- azp(nc, attrs = c("SID74", "SID79"), n_regions = 8,
              method = "tabu", tabu_length = 15)

# Simulated annealing
result <- azp(nc, attrs = c("SID74", "SID79"), n_regions = 8,
              method = "sa", cooling_rate = 0.95)

# View results
plot(result[".region"])
```

---

cflp *Capacitated Facility Location Problem (CFLP)*

Description

Solves the Capacitated Facility Location Problem: minimize total weighted distance from demand points to facilities, subject to capacity constraints at each facility. Unlike standard p-median, facilities have limited capacity and demand may need to be split across multiple facilities.

Usage

```
cflp(
  demand,
  facilities,
  n_facilities,
  weight_col,
  capacity_col,
  facility_cost_col = NULL,
  cost_matrix = NULL,
  distance_metric = "euclidean",
  verbose = FALSE
)
```

Arguments

demand	An sf object representing demand points.
facilities	An sf object representing candidate facility locations.

n_facilities	Integer. Number of facilities to locate. Set to 0 if using facility_cost_col to determine optimal number.
weight_col	Character. Column name in demand containing demand weights (e.g., population, customers, volume).
capacity_col	Character. Column name in facilities containing capacity of each facility.
facility_cost_col	Optional character. Column name in facilities containing fixed cost to open each facility. If provided and n_facilities = 0, the solver determines the optimal number of facilities to minimize total cost.
cost_matrix	Optional. Pre-computed distance/cost matrix (demand x facilities).
distance_metric	Distance metric: "euclidean" (default) or "manhattan".
verbose	Logical. Print solver progress.

Details

The CFLP extends the p-median problem by adding capacity constraints. Each facility j has a maximum capacity Q_j, and the total demand assigned to it cannot exceed this capacity.

When demand exceeds available capacity at the nearest facility, the solver may split demand across multiple facilities. The .split column indicates which demand points have been split, and the allocation_matrix in metadata shows the exact fractions.

Two modes of operation:

1. **Fixed number**: Set n_facilities to select exactly that many facilities
2. **Cost-based**: Set n_facilities = 0 and provide facility_cost_col to let the solver determine the optimal number based on fixed + variable costs

Value

A list with two sf objects:

- $demand: Original demand sf with .facility column (primary assignment) and .split column (TRUE if demand is split across facilities)
- $facilities: Original facilities sf with .selected, .n_assigned, and .utilization columns

Metadata is stored in the "spopt" attribute, including:

- objective: Total cost (transportation + facility costs if applicable)
- mean_distance: Mean weighted distance
- n_split_demand: Number of demand points split across facilities
- allocation_matrix: Full allocation matrix (n_demand x n_facilities)

References

Daskin, M. S. (2013). Network and discrete location: Models, algorithms, and applications (2nd ed.). John Wiley & Sons. doi:10.1002/9781118537015

Sridharan, R. (1995). The capacitated plant location problem. European Journal of Operational Research, 87(2), 203-213. doi:10.1016/03772217(95)00042O

Examples

```
library(sf)

# Demand points with population
demand <- st_as_sf(data.frame(
  x = runif(100), y = runif(100), population = rpois(100, 500)
), coords = c("x", "y"))

# Facilities with varying capacities
facilities <- st_as_sf(data.frame(
  x = runif(15), y = runif(15),
  capacity = c(rep(5000, 5), rep(10000, 5), rep(20000, 5)),
  fixed_cost = c(rep(100, 5), rep(200, 5), rep(400, 5))
), coords = c("x", "y"))

# Fixed number of facilities
result <- cflp(demand, facilities, n_facilities = 5,
               weight_col = "population", capacity_col = "capacity")

# Check utilization
result$facilities[result$facilities$.selected, c("capacity", ".utilization")]

# Cost-based (optimal number of facilities)
result <- cflp(demand, facilities, n_facilities = 0,
               weight_col = "population", capacity_col = "capacity",
               facility_cost_col = "fixed_cost")
attr(result, "spopt")$n_selected
```

`corridor_graph` *Build a Corridor Graph for Cached Routing*

Description

Pre-build the routing graph from a cost surface so that multiple `route_corridor` calls can skip graph construction. The returned object is a snapshot of the cost surface at build time; subsequent edits to the raster do not affect the graph.

Usage

```
corridor_graph(cost_surface, neighbours = 8L, resolution_factor = 1L)
```

Arguments

`cost_surface`	A terra SpatRaster (single band). Same requirements as `route_corridor`.
`neighbours`	Integer. Cell connectivity: 4, 8 (default), or 16. Fixed at build time.
`resolution_factor`	Integer, default 1L. If > 1, the surface is aggregated before graph construction. Fixed at build time.

Details

The object retains both the CSR graph (in Rust) and an independent copy of the raster (for coordinate mapping). The printed `graph_storage` reflects only the CSR arrays, not the raster copy.

Value

An opaque `spopt_corridor_graph` object for use with `route_corridor`.

Examples

```
library(terra)
r <- rast(nrows = 500, ncols = 500, xmin = 0, xmax = 500000,
          ymin = 0, ymax = 500000, crs = "EPSG:32614")
values(r) <- runif(ncell(r), 0.5, 2.0)

g <- corridor_graph(r, neighbours = 8L)
print(g)
path <- route_corridor(g, c(50000, 50000), c(450000, 450000))
```

`delivery_data` *Delivery routing data for Fort Worth, TX*

Description

Pre-computed delivery stop locations and travel-time matrix for a route optimization scenario in western Tarrant and Parker counties, Texas. The depot is an Amazon delivery station on San Jacinto Dr in Fort Worth, with 25 residential delivery stops in Fort Worth, Benbrook, Aledo, and Willow Park.

Usage

```
delivery_data
```

Format

A list with four elements:

stops An sf object with 26 rows (1 depot + 25 delivery stops) and columns:

- **id** Stop identifier ("depot", "D01", ..., "D25")
- **address** Street address
- **packages** Number of packages for delivery (0 for depot)

matrix A 26x26 numeric matrix of driving travel times in minutes, computed with r5r using OpenStreetMap road network data. Row and column names correspond to stop IDs.

tsp_route An sf LINESTRING object with road-snapped route geometries for the optimized single-vehicle tour, one row per leg.

vrp_route An sf LINESTRING object with road-snapped route geometries for the optimized multi-vehicle solution, with a `vehicle` column indicating which van each leg belongs to.

Source

Travel times computed with r5r v2.3 using OpenStreetMap data for Tarrant and Parker counties, Texas. Addresses geocoded with Nominatim via tidygeocoder.

`distance_matrix` *Compute distance matrix between sf objects*

Description

Computes pairwise distances between geometries. For geographic (longlat) coordinate systems, uses great circle distances in meters via `sf::st_distance()`. For projected coordinate systems, uses fast Euclidean distance in the CRS units (typically meters).

Usage

```
distance_matrix(
  x,
  y = NULL,
  type = c("euclidean", "manhattan"),
  use_centroids = NULL
)
```

Arguments

`x`	An sf object (demand points for facility location, or areas for regionalization).
`y`	An sf object (facility locations). If NULL, computes distances within x.
`type`	Distance type: "euclidean" (default) or "manhattan". Note that for geographic CRS, only "euclidean" (great circle) distance is available.
`use_centroids`	Logical. If TRUE (default for polygons), use polygon centroids.

Value

A numeric matrix of distances. Rows correspond to x, columns to y. For geographic CRS, distances are in meters. For projected CRS, distances are in the CRS units (usually meters).

Examples

```
library(sf)
demand <- st_as_sf(data.frame(x = runif(10), y = runif(10)), coords = c("x", "y"))
facilities <- st_as_sf(data.frame(x = runif(5), y = runif(5)), coords = c("x", "y"))
d <- distance_matrix(demand, facilities)
```

`frlm` *Flow Refueling Location Model (FRLM)*

Description

Solves the Flow Refueling Location Model to optimally place refueling facilities along network paths. This model maximizes the volume of origin-destination flows that can be served given vehicle range constraints.

Usage

```
frlm(
  flows,
  candidates,
  network = NULL,
  vehicle_range,
  n_facilities,
  method = c("greedy"),
  verbose = FALSE
)
```

Arguments

`flows`	A data frame or sf object containing flow information with columns: • `origin`: Origin identifier • `destination`: Destination identifier • `volume`: Flow volume (e.g., number of trips)
`candidates`	An sf object with candidate facility locations (points).
`network`	Optional. A distance matrix between candidates. If NULL (default), Euclidean distances are computed from candidate geometries. For network distances, compute externally using packages like r5r or dodgr and pass the resulting matrix here.
`vehicle_range`	Numeric. Maximum vehicle range (same units as network distances).
`n_facilities`	Integer. Number of facilities to place.
`method`	Character. Optimization method: "greedy" (default and currently only option).
`verbose`	Logical. Print progress messages.

Details

The Flow Refueling Location Model (Kuby & Lim, 2005) addresses the problem of locating refueling stations for range-limited vehicles (e.g., electric vehicles, hydrogen fuel cell vehicles) along travel paths.

A flow (origin-destination path) is "covered" if a vehicle can complete the **round trip** with refueling stops at the selected facilities. The model assumes:

- Vehicles start at the origin with **half a tank** (can travel R/2)
- At each open station, vehicles refuel to full (can travel R)
- The round trip must be completable without running out of fuel

For a flow to be covered, three conditions must be met:

1. First open station must be within R/2 from origin (half-tank start)
2. Each subsequent open station must be within R of the previous
3. Last open station must be within R/2 of destination (to allow return)

This implementation uses a greedy heuristic that iteratively selects the facility providing the greatest marginal increase in covered flow volume.

Value

A list with class "spopt_frlm" containing:

- `facilities`: The candidates sf object with a `.selected` column
- `selected_indices`: 1-based indices of selected facilities
- `coverage`: Coverage statistics

Metadata is stored in the "spopt" attribute.

Input Format

For simple cases, you can provide:

- `flows`: Data frame with origin, destination, volume
- `candidates`: sf points for potential facility locations
- `network`: Pre-computed distance matrix (optional)

References

Kuby, M., & Lim, S. (2005). The flow-refueling location problem for alternative-fuel vehicles. Socio-Economic Planning Sciences, 39(2), 125-145. doi:10.1016/j.seps.2004.03.001

Capar, I., & Kuby, M. (2012). An efficient formulation of the flow refueling location model for alternative-fuel stations. IIE Transactions, 44(8), 622-636. doi:10.1080/0740817X.2011.635175

Examples

```
# Simple example with distance matrix
library(sf)

# Create candidate locations
candidates <- st_as_sf(data.frame(
  id = 1:10,
  x = runif(10, 0, 100),
  y = runif(10, 0, 100)
), coords = c("x", "y"))
```

```
# Create flows (using candidate indices as origins/destinations)
flows <- data.frame(
  origin = c(1, 1, 3, 5),
  destination = c(8, 10, 7, 9),
  volume = c(100, 200, 150, 300)
)

# Solve with vehicle range of 50 units
result <- frlm(flows, candidates, vehicle_range = 50, n_facilities = 3)

# View selected facilities
result$facilities[result$facilities$.selected, ]
```

---

huff *Huff Model for Market Share Analysis*

Description

Computes probability surfaces to predict market share and sales potential based on distance decay and store attractiveness. The Huff model is widely used in retail site selection to estimate the probability that a consumer at a given location will choose a particular store.

Usage

```
huff(
  demand,
  stores,
  attractiveness_col,
  attractiveness_exponent = 1,
  distance_exponent = -1.5,
  sales_potential_col = NULL,
  cost_matrix = NULL,
  distance_metric = "euclidean"
)
```

Arguments

demand	An sf object representing demand points or areas. Can be customer locations, census block groups, grid cells, etc.
stores	An sf object representing store/facility locations.
attractiveness_col	Character vector. Column name(s) in `stores` containing attractiveness values (e.g., square footage, parking spaces). Multiple columns can be specified for composite attractiveness.

`attractiveness_exponent`
: Numeric vector. Exponent(s) for attractiveness (default 1). Must be same length as `attractiveness_col` or length 1 (recycled). Higher values increase the importance of that variable.

`distance_exponent`
: Numeric. Distance decay exponent (default -1.5).

Should be negative; more negative = faster decay with distance.

`sales_potential_col`
: Optional character. Column name in `demand` containing sales potential values (e.g., disposable income, population). If NULL, each demand point is weighted equally.

`cost_matrix`
: Optional. Pre-computed distance/cost matrix (demand x stores). If NULL, Euclidean distance is computed from geometries.

`distance_metric`
: Distance metric if cost_matrix is NULL: "euclidean" (default) or "manhattan".

Details

The Huff model calculates the probability that a consumer at location i will choose store j using:

$$P_{ij} = \frac{A_j \times D_{ij}^{\beta}}{\sum_k A_k \times D_{ik}^{\beta}}$$

Where:

- A_j is the composite attractiveness of store j
- D_{ij} is the distance from i to j
- β is the distance decay exponent (default -1.5)

When multiple attractiveness variables are specified, the composite attractiveness is computed as:

$$A_j = \prod_m V_{jm}^{\alpha_m}$$

Where V_{jm} is the value of attractiveness variable m for store j, and α_m is the corresponding exponent.

The distance exponent is typically negative because probability decreases with distance. Common values range from -1 to -3.

Value

A list with:

- `$demand`: Original demand sf with added columns:
  - `.primary_store`: ID of highest-probability store
  - `.entropy`: Competition measure (higher = more competition)
  - `.prob_<store_id>`: Probability columns for each store

- `$stores`: Original stores sf with added columns:
  - `.market_share`: Proportion of total market captured
  - `.expected_sales`: Expected sales (sum of prob × potential)
- `$probability_matrix`: Full probability matrix (n_demand × n_stores)

Metadata in "spopt" attribute includes parameters used.

Outputs

Market Share: The weighted average probability across all demand points, representing the proportion of total market potential captured by each store.

Expected Sales: The sum of (probability × sales_potential) for each store, representing the expected sales volume.

Entropy: A measure of local competition. Higher entropy indicates more competitive areas where multiple stores have similar probabilities.

References

Huff, D. L. (1963). A Probabilistic Analysis of Shopping Center Trade Areas. Land Economics, 39(1), 81-90. doi:10.2307/3144521

Huff, D. L. (1964). Defining and Estimating a Trading Area. Journal of Marketing, 28(3), 34-38. doi:10.1177/002224296402800307

Examples

```
library(sf)

# Create demand grid with spending potential
demand <- st_as_sf(expand.grid(x = 1:10, y = 1:10), coords = c("x", "y"))
demand$spending <- runif(100, 1000, 5000)

# Existing stores with varying sizes (attractiveness)
stores <- st_as_sf(data.frame(
  id = c("Store_A", "Store_B", "Store_C"),
  sqft = c(50000, 25000, 75000),
  parking = c(200, 100, 300),
  x = c(2, 8, 5), y = c(2, 8, 5)
), coords = c("x", "y"))

# Single attractiveness variable
result <- huff(demand, stores,
               attractiveness_col = "sqft",
               distance_exponent = -2,
               sales_potential_col = "spending")

# Multiple attractiveness variables with different exponents
# Composite: A = sqft^1.0 * parking^0.5
result_multi <- huff(demand, stores,
                     attractiveness_col = c("sqft", "parking"),
                     attractiveness_exponent = c(1.0, 0.5),
```

```
                    distance_exponent = -2,
                    sales_potential_col = "spending")

# View market shares
result_multi$stores[, c("id", "sqft", "parking", ".market_share", ".expected_sales")]

# Evaluate a new candidate store
candidate <- st_as_sf(data.frame(
  id = "New_Store", sqft = 40000, parking = 250, x = 3, y = 7
), coords = c("x", "y"))

all_stores <- rbind(stores, candidate)
result_with_candidate <- huff(demand, all_stores,
                              attractiveness_col = c("sqft", "parking"),
                              attractiveness_exponent = c(1.0, 0.5),
                              distance_exponent = -2,
                              sales_potential_col = "spending")

# Compare market shares with and without candidate
result_with_candidate$stores[, c("id", ".market_share")]
```

`lscp` *Location Set Covering Problem (LSCP)*

Description

Solves the Location Set Covering Problem: find the minimum number of facilities needed to cover all demand points within a given service radius.

Usage

```
lscp(
  demand,
  facilities,
  service_radius,
  cost_matrix = NULL,
  distance_metric = "euclidean",
  verbose = FALSE
)
```

Arguments

`demand`	An sf object representing demand points (or polygons, using centroids).
`facilities`	An sf object representing candidate facility locations.
`service_radius`	Numeric. Maximum distance for a facility to cover a demand point.

`cost_matrix` Optional. Pre-computed distance matrix (demand x facilities). If NULL, computed from geometries.

`distance_metric` Distance metric: "euclidean" (default) or "manhattan".

`verbose` Logical. Print solver progress.

Details

The LSCP minimizes the number of facilities required to ensure that every demand point is within the service radius of at least one facility. This is a mandatory coverage model where full coverage is required.

The integer programming formulation is:

$$\min \sum_j y_j$$

Subject to:

$$\sum_j a_{ij} y_j \geq 1 \quad \forall i$$

$$y_j \in \{0, 1\}$$

Where $y_j = 1$ if facility j is selected, and $a_{ij} = 1$ if facility j can cover demand point i (distance $\leq$ service radius).

Value

A list with two sf objects:

- `$demand`: Original demand sf with `.covered` column (logical)
- `$facilities`: Original facilities sf with `.selected` column (logical)

Metadata is stored in the "spopt" attribute.

Use Cases

LSCP is appropriate when complete coverage is mandatory:

- **Emergency services**: Fire stations, ambulance depots, or hospitals where every resident must be reachable within a response time standard
- **Public services**: Schools, polling places, or post offices where universal access is required by law or policy
- **Infrastructure**: Cell towers or utility substations where gaps in coverage are unacceptable
- **Retail/logistics**: Warehouse locations to ensure all customers can receive same-day or next-day delivery

For situations where complete coverage is not required or not feasible within budget constraints, consider `mclp()` instead.

References

Toregas, C., Swain, R., ReVelle, C., & Bergman, L. (1971). The Location of Emergency Service Facilities. Operations Research, 19(6), 1363-1373. doi:10.1287/opre.19.6.1363

See Also

`mclp()` for maximizing coverage with a fixed number of facilities

Examples

```
library(sf)

# Create demand and facility points
demand <- st_as_sf(data.frame(x = runif(50), y = runif(50)), coords = c("x", "y"))
facilities <- st_as_sf(data.frame(x = runif(10), y = runif(10)), coords = c("x", "y"))

# Find minimum facilities to cover all demand within 0.3 units
result <- lscp(demand, facilities, service_radius = 0.3)

# View selected facilities
result$facilities[result$facilities$.selected, ]
```

---

max_p_regions *Max-P Regions*

Description

Perform Max-P regionalization to maximize the number of spatially contiguous regions such that each region satisfies a minimum threshold constraint on a specified attribute. This is useful for creating regions that meet minimum population or sample size requirements.

Usage

```
max_p_regions(
  data,
  attrs = NULL,
  threshold_var,
  threshold,
  weights = "queen",
  bridge_islands = FALSE,
  compact = FALSE,
  compact_weight = 0.5,
  compact_metric = "centroid",
  homogeneous = TRUE,
  n_iterations = 100L,
  n_sa_iterations = 100L,
```

```
  cooling_rate = 0.99,
  tabu_length = 10L,
  scale = TRUE,
  seed = NULL,
  verbose = FALSE
)
```

Arguments

data	An sf object with polygon or point geometries.
attrs	Character vector of column names to use for computing within-region dissimilarity (e.g., c("var1", "var2")). If NULL, uses all numeric columns.
threshold_var	Character. Name of the column containing the threshold variable (e.g., population, income). Each region must have a sum of this variable >= threshold.
threshold	Numeric. Minimum sum of threshold_var required per region.
weights	Spatial weights specification. Can be: • "queen" (default): Polygons sharing any boundary point are neighbors • "rook": Polygons sharing an edge are neighbors • An nb object from spdep or created with sp_weights() • A list for other weight types: list(type = "knn", k = 6) for k-nearest neighbors, or list(type = "distance", d = 5000) for distance-based weights KNN weights guarantee connectivity (no islands), which can be useful for datasets with disconnected polygons.
bridge_islands	Logical. If TRUE, automatically connect disconnected components (e.g., islands) using nearest-neighbor edges. If FALSE (default), the function will error when the spatial weights graph is disconnected. This is useful for datasets like LA County with Catalina Islands, or archipelago data where physical adjacency doesn't exist but regionalization is still desired.
compact	Logical. If TRUE, optimize for region compactness in addition to attribute homogeneity. Compact regions have more regular shapes, which is useful for sales territories, patrol areas, and electoral districts. Default is FALSE.
compact_weight	Numeric between 0 and 1. Weight for compactness vs attribute homogeneity when compact = TRUE. Higher values prioritize compact shapes over attribute similarity. Default is 0.5.
compact_metric	Either "centroid" (the default) or "nmi" (Normalized Moment of Inertia).
homogeneous	Logical. If TRUE, minimizes within-region dissimilarity, so that regions are internally homogeneous. If FALSE, maximizes within-region dissimilarity.
n_iterations	Integer. Number of construction phase iterations (default 100). Higher values explore more random starting solutions.
n_sa_iterations	Integer. Number of simulated annealing iterations (default 100). Set to 0 to skip the SA refinement phase.
cooling_rate	Numeric. SA cooling rate between 0 and 1 (default 0.99). Smaller values cool faster, larger values allow more exploration.

tabu_length	Integer. Length of tabu list for SA phase (default 10).
scale	Logical. If TRUE (default), standardize attributes before computing dissimilarity.
seed	Optional integer for reproducibility.
verbose	Logical. Print progress messages.

Details

The Max-P algorithm (Duque, Anselin & Rey, 2012; Wei, Rey & Knaap, 2021) solves the problem of aggregating n geographic areas into the maximum number of homogeneous regions while ensuring:

1. Each region is spatially contiguous (connected)
2. Each region satisfies a minimum threshold on a specified attribute

The algorithm has two phases:

1. Construction phase: Builds feasible solutions via randomized greedy region growing. Multiple random starts are explored in parallel.
2. Simulated annealing phase: Refines solutions by moving border areas between regions to minimize within-region dissimilarity while respecting constraints.

When `compact = TRUE`, the algorithm additionally optimizes for compact region shapes based on Feng, Rey, & Wei (2022). Compact regions:

- Minimize travel time within regions (useful for service territories)
- Reduce gerrymandering potential (electoral districts)
- Often result in finding MORE regions due to efficient space usage

Compactness metric: This implementation provides two options for the compactness metric used during optimization. The Normalized Moment of Inertia (NMI) described in Feng et al. (2022) can be used. However, the default option is a dispersion measure. The dispersion measure has two advantages:

1. **Point-based regionalization**: The algorithm works with both polygon and point geometries. For point data, use KNN or distance-based weights (e.g., `weights = list(type = "knn", k = 6)`).
2. **Computational efficiency**: Centroid dispersion is O(n) per region versus O(v) for NMI where v = total polygon vertices.

For polygon data, centroids are computed via `sf::st_centroid()`. Users should be aware that centroid-based compactness may be less accurate for highly irregular shapes or large, sparsely-populated areas where the centroid poorly represents the polygon's spatial extent.

The reported `mean_compactness` and `region_compactness` in results use Polsby-Popper (4*pi*A/P^2), a standard geometric compactness measure for polygons. For point data, these metrics are not computed.

This implementation is optimized for speed using:

- Parallel construction with early termination
- Efficient articulation point detection for move eligibility
- Incremental threshold tracking

Value

An sf object with a `.region` column containing region assignments. Metadata is stored in the "spopt" attribute, including:

- algorithm: "max_p"
- n_regions: Number of regions created (the "p" in max-p)
- objective: Total within-region sum of squared deviations
- threshold_var: Name of threshold variable
- threshold: Threshold value used
- solve_time: Time to solve in seconds
- mean_compactness: Mean Polsby-Popper compactness (if `compact = TRUE`)
- region_compactness: Per-region compactness scores (if `compact = TRUE`)

References

Duque, J. C., Anselin, L., & Rey, S. J. (2012). The max-p-regions problem. Journal of Regional Science, 52(3), 397-419.

Wei, R., Rey, S., & Knaap, E. (2021). Efficient regionalization for spatially explicit neighborhood delineation. International Journal of Geographical Information Science, 35(1), 135-151. doi:10.1080/13658816.2020.1759806

Feng, X., Rey, S., & Wei, R. (2022). The max-p-compact-regions problem. Transactions in GIS, 26, 717-734. doi:10.1111/tgis.12874

Examples

```
library(sf)
nc <- st_read(system.file("shape/nc.shp", package = "sf"))

# Create regions where each has at least 100,000 in BIR74
result <- max_p_regions(
  nc,
  attrs = c("SID74", "SID79"),
  threshold_var = "BIR74",
  threshold = 100000
)

# Check number of regions created
attr(result, "spopt")$n_regions

# With compactness optimization (for sales territories)
result_compact <- max_p_regions(
  nc,
  attrs = c("SID74", "SID79"),
  threshold_var = "BIR74",
  threshold = 100000,
  compact = TRUE,
  compact_weight = 0.5
)
```

```
# Check compactness
attr(result_compact, "spopt")$mean_compactness

# Plot results
plot(result[".region"])

# Point-based regionalization (e.g., store locations, sensor networks)
# Use KNN weights since points don't have polygon contiguity
points <- st_as_sf(data.frame(
  x = runif(200), y = runif(200),
  customers = rpois(200, 100),
  avg_income = rnorm(200, 50000, 15000)
), coords = c("x", "y"))

result_points <- max_p_regions(
  points,
  attrs = "avg_income",
  threshold_var = "customers",
  threshold = 500,
  weights = list(type = "knn", k = 6),
  compact = TRUE
)
```

---

mclp *Maximum Coverage Location Problem (MCLP)*

Description

Solves the Maximum Coverage Location Problem: maximize total weighted demand covered by locating exactly p facilities.

Usage

```
mclp(
  demand,
  facilities,
  service_radius,
  n_facilities,
  weight_col,
  cost_matrix = NULL,
  distance_metric = "euclidean",
  fixed_col = NULL,
  verbose = FALSE
)
```

Arguments

demand	An sf object representing demand points.
facilities	An sf object representing candidate facility locations.
service_radius	Numeric. Maximum distance for coverage.
n_facilities	Integer. Number of facilities to locate (p).
weight_col	Character. Column name in demand containing demand weights.
cost_matrix	Optional. Pre-computed distance matrix.
distance_metric	Distance metric: "euclidean" (default) or "manhattan".
fixed_col	Optional column name in facilities indicating which facilities are pre-selected. The column should be logical (TRUE for fixed) or character ("required"/"candidate"). Fixed facilities are always selected; the solver optimizes the remaining slots. Useful for expansion planning where some facilities already exist.
verbose	Logical. Print solver progress.

Details

The MCLP maximizes the total weighted demand covered by locating exactly p facilities. Unlike lscp(), which requires full coverage, MCLP accepts partial coverage and optimizes for the best possible outcome given a fixed budget (number of facilities).

The integer programming formulation is:

$$\max \sum_i w_i z_i$$

Subject to:

$$\sum_j y_j = p$$

$$z_i \leq \sum_j a_{ij} y_j \quad \forall i$$

$$y_j, z_i \in \{0, 1\}$$

Where w_i is the weight (demand) at location i, $y_j = 1$ if facility j is selected, $z_i = 1$ if demand i is covered, and $a_{ij} = 1$ if facility j can cover demand i.

Value

A list with two sf objects:

- $demand: Original demand sf with .covered and .facility columns
- $facilities: Original facilities sf with .selected column

Metadata is stored in the "spopt" attribute.

Use Cases

MCLP is appropriate when you have a fixed budget or capacity constraint:

- **Healthcare access**: Locating p clinics to maximize the population within a 30-minute drive, given limited funding
- **Retail site selection**: Choosing p store locations to maximize the number of potential customers within a trade area
- **Emergency services**: Placing p ambulance stations to maximize the population reachable within an 8-minute response time
- **Conservation**: Selecting p reserve sites to maximize the number of species or habitat area protected
- **Telecommunications**: Locating p cell towers to maximize population coverage when full coverage is not economically feasible

For situations where complete coverage is required, use `lscp()` to find the minimum number of facilities needed.

References

Church, R., & ReVelle, C. (1974). The Maximal Covering Location Problem. Papers in Regional Science, 32(1), 101-118. doi:10.1007/BF01942293

See Also

`lscp()` for finding the minimum facilities needed for complete coverage

Examples

```
library(sf)

# Create demand with weights
demand <- st_as_sf(data.frame(
  x = runif(50), y = runif(50), population = rpois(50, 100)
), coords = c("x", "y"))
facilities <- st_as_sf(data.frame(x = runif(10), y = runif(10)), coords = c("x", "y"))

# Maximize population coverage with 3 facilities
result <- mclp(demand, facilities, service_radius = 0.3,
               n_facilities = 3, weight_col = "population")

attr(result, "spopt")$coverage_pct
```

nmi *Normalized moment of inertia (NMI)*

Description

Computes the normalized moment of inertia (NMI), a compactness measure for polygon geometries. The NMI ranges between 0 and 1, where 1 is the most compact shape (a circle) and 0 is an infinitely extending shape (Feng et al. 2022).

Usage

```
nmi(x)
```

Arguments

x	An sf object, sfc geometry column, or sfg geometry

Details

The NMI is defined as follows, where A is the area of a geometry, and I is the second moment of inertia (i.e., the second areal moment):

$$\frac{A^2}{2\pi I}$$

See Li et al. (2013, 2014) for additional details.

Value

Numeric vector of normalized moments of inertia.

References

Feng, X., Rey, S., and Wei, R. (2022). "The max-p-compact-regions problem." Transactions in GIS, 26, 717–734. doi:10.1111/tgis.12874.

Li, W., Goodchild, M.F., and Church, R.L. 2013. "An Efficient Measure of Compactness for Two-Dimensional Shapes and Its Application in Regionalization Problems." International Journal of Geographical Information Science 27 (6): 1227–50. doi:10.1080/13658816.2012.752093.

Li, W., Church, R.L. and Goodchild, M.F. 2014. "The p-Compact-regions Problem." Geogr Anal, 46: 250-273. doi:10.1111/gean.12038.

See Also

second_areal_moment()

plot.spopt_k_corridors *Plot k-Diverse Corridors*

Description

Renders all k corridors on a single plot. Rank 1 is drawn with full opacity and maximum line width; higher ranks fade and thin proportionally.

Usage

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

Arguments

x	An spopt_k_corridors object from route_k_corridors.
...	Additional arguments passed to the initial plot() call.

Value

No return value, called for side effects (draws the corridors to the active graphics device).

p_center *P-Center Problem*

Description

Solves the P-Center problem: minimize the maximum distance from any demand point to its nearest facility by locating exactly p facilities. This is an equity-focused (minimax) objective that ensures no demand point is too far from service.

Usage

```
p_center(
  demand,
  facilities,
  n_facilities,
  cost_matrix = NULL,
  distance_metric = "euclidean",
  method = c("binary_search", "mip"),
  fixed_col = NULL,
  verbose = FALSE
)
```

Arguments

`demand`	An sf object representing demand points.
`facilities`	An sf object representing candidate facility locations.
`n_facilities`	Integer. Number of facilities to locate (p).
`cost_matrix`	Optional. Pre-computed distance matrix.
`distance_metric`	Distance metric: "euclidean" (default) or "manhattan".
`method`	Algorithm to use: "binary_search" (default, faster) or "mip" (direct mixed-integer programming formulation).
`fixed_col`	Optional column name in `facilities` indicating which facilities are pre-selected. The column should be logical (`TRUE` for fixed) or character (`"required"`/`"candidate"`). Fixed facilities are always selected; the solver optimizes the remaining slots.
`verbose`	Logical. Print solver progress.

Details

The p-center problem minimizes the maximum distance between any demand point and its assigned facility. This "minimax" objective ensures equitable access by focusing on the worst-served location rather than average performance.

Two algorithms are available:

- `"binary_search"` (default): Binary search over distances with set covering subproblems. This converts the difficult minimax objective into simpler feasibility problems and is typically much faster.
- `"mip"`: Direct mixed-integer programming formulation with the minimax objective. Can be slower but may be preferred for small problems or when exact optimality certificates are needed.

The direct MIP formulation is:

$$\min W$$

Subject to:

$$\sum_j y_j = p$$

$$\sum_j x_{ij} = 1 \quad \forall i$$

$$x_{ij} \leq y_j \quad \forall i, j$$

$$\sum_j d_{ij} x_{ij} \leq W \quad \forall i$$

$$x_{ij}, y_j \in \{0, 1\}$$

Where W is the maximum distance to minimize, d_{ij} is the distance from demand i to facility j, $x_{ij} = 1$ if demand i is assigned to facility j, and $y_j = 1$ if facility j is selected.

Value

A list with two sf objects:

- `$demand`: Original demand sf with `.facility` column
- `$facilities`: Original facilities sf with `.selected` column

Metadata includes `max_distance` (the objective value).

Use Cases

P-center is appropriate when equity and worst-case performance matter:

- **Emergency services**: Fire stations or ambulance depots where response time standards must be met for all residents
- **Equity-focused planning**: Ensuring no community is underserved, even if it increases average travel distance
- **Critical infrastructure**: Backup facilities or emergency shelters where everyone must be within reach
- **Service level guarantees**: When contracts or regulations specify maximum acceptable distance or response time

For efficiency-focused objectives that minimize total travel, consider `p_median()` instead.

References

Hakimi, S. L. (1965). Optimum Distribution of Switching Centers in a Communication Network and Some Related Graph Theoretic Problems. Operations Research, 13(3), 462-475. doi:10.1287/opre.13.3.462

See Also

`p_median()` for minimizing total weighted distance (efficiency objective)

Examples

```
library(sf)

demand <- st_as_sf(data.frame(x = runif(50), y = runif(50)), coords = c("x", "y"))
facilities <- st_as_sf(data.frame(x = runif(15), y = runif(15)), coords = c("x", "y"))

# Minimize maximum distance with 4 facilities
result <- p_center(demand, facilities, n_facilities = 4)

# Maximum distance any demand point must travel
attr(result, "spopt")$max_distance
```

`p_dispersion` *P-Dispersion Problem*

Description

Solves the P-Dispersion problem: maximize the minimum distance between any two selected facilities. This "maximin" objective ensures facilities are spread out as much as possible.

Usage

```
p_dispersion(
  facilities,
  n_facilities,
  cost_matrix = NULL,
  distance_metric = "euclidean",
  verbose = FALSE
)
```

Arguments

`facilities`	An sf object representing candidate facility locations. Note: This problem does not use demand points.
`n_facilities`	Integer. Number of facilities to locate (p).
`cost_matrix`	Optional. Pre-computed inter-facility distance matrix.
`distance_metric`	Distance metric: "euclidean" (default) or "manhattan".
`verbose`	Logical. Print solver progress.

Details

The p-dispersion problem selects p facilities from a set of candidates such that the minimum pairwise distance between any two selected facilities is maximized. Unlike p-median or p-center, this problem does not consider demand points—it focuses solely on spreading facilities apart.

The mixed integer programming formulation uses a Big-M approach:

$$\max D$$

Subject to:

$$\sum_j y_j = p$$

$$D \leq d_{ij} + M(2 - y_i - y_j) \quad \forall i < j$$

$$y_j \in \{0, 1\}, \quad D \geq 0$$

Where D is the minimum separation distance to maximize, d_{ij} is the distance between facilities i and j, $y_j = 1$ if facility j is selected, and M is a large constant. When both facilities i and j are selected ($y_i = y_j = 1$), the constraint reduces to $D \leq d_{ij}$, ensuring D is at most the distance between any pair of selected facilities.

Value

An sf object (the facilities input) with a `.selected` column. Metadata includes `min_distance` (the objective value).

Use Cases

P-dispersion is appropriate when facilities should be spread apart:

- **Obnoxious facilities**: Hazardous waste sites, prisons, or other undesirable facilities that should be separated from each other
- **Franchise territories**: Retail locations where stores should not cannibalize each other's market
- **Redundant systems**: Backup servers or emergency caches that should be geographically distributed for resilience
- **Monitoring networks**: Air quality sensors or seismic monitors that should cover distinct areas without overlap
- **Spatial sampling**: Selecting representative sample locations that are well-distributed across a study area

References

Kuby, M. J. (1987). Programming Models for Facility Dispersion: The p-Dispersion and Maxisum Dispersion Problems. Geographical Analysis, 19(4), 315-329. doi:10.1111/j.15384632.1987.tb00133.x

Examples

```
library(sf)

facilities <- st_as_sf(data.frame(x = runif(20), y = runif(20)), coords = c("x", "y"))

# Select 5 facilities maximally dispersed
result <- p_dispersion(facilities, n_facilities = 5)

# Minimum distance between any two selected facilities
attr(result, "spopt")$min_distance
```

---

p_median *P-Median Problem*

Description

Solves the P-Median problem: minimize total weighted distance from demand points to their assigned facilities by locating exactly p facilities. This is an efficiency-focused objective that minimizes overall travel burden.

Usage

```
p_median(
  demand,
  facilities,
  n_facilities,
  weight_col,
  cost_matrix = NULL,
  distance_metric = "euclidean",
  fixed_col = NULL,
  verbose = FALSE
)
```

Arguments

demand	An sf object representing demand points.
facilities	An sf object representing candidate facility locations.
n_facilities	Integer. Number of facilities to locate (p).
weight_col	Character. Column name in demand containing demand weights.
cost_matrix	Optional. Pre-computed distance matrix.
distance_metric	Distance metric: "euclidean" (default) or "manhattan".
fixed_col	Optional column name in facilities indicating which facilities are pre-selected. The column should be logical (TRUE for fixed) or character ("required"/"candidate"). Fixed facilities are always selected; the solver optimizes the remaining slots.
verbose	Logical. Print solver progress.

Details

The p-median problem minimizes the total weighted distance (or travel cost) between demand points and their nearest assigned facility. It is the most widely used location model for efficiency-oriented facility siting.

The integer programming formulation is:

$$\min \sum_i \sum_j w_i d_{ij} x_{ij}$$

Subject to:

$$\sum_j y_j = p$$

$$\sum_j x_{ij} = 1 \quad \forall i$$

$$x_{ij} \leq y_j \quad \forall i, j$$

$$x_{ij}, y_j \in \{0, 1\}$$

Where w_i is the demand weight at location i, d_{ij} is the distance from demand i to facility j, $x_{ij} = 1$ if demand i is assigned to facility j, and $y_j = 1$ if facility j is selected.

Value

A list with two sf objects:

- `$demand`: Original demand sf with `.facility` column (assigned facility)
- `$facilities`: Original facilities sf with `.selected` and `.n_assigned` columns

Metadata is stored in the "spopt" attribute.

Use Cases

P-median is appropriate when minimizing total travel cost or distance:

- **Public facilities**: Schools, libraries, or community centers where the goal is to minimize total student/patron travel
- **Warehouses and distribution**: Locating distribution centers to minimize total shipping costs to customers
- **Healthcare**: Positioning clinics to minimize aggregate patient travel time across a population
- **Service depots**: Locating maintenance facilities to minimize total technician travel to service calls

For equity-focused objectives where no demand point should be too far, consider `p_center()` instead.

References

Hakimi, S. L. (1964). Optimum Locations of Switching Centers and the Absolute Centers and Medians of a Graph. Operations Research, 12(3), 450-459. doi:10.1287/opre.12.3.450

See Also

`p_center()` for minimizing maximum distance (equity objective)

Examples

```
library(sf)

demand <- st_as_sf(data.frame(
  x = runif(100), y = runif(100), population = rpois(100, 500)
), coords = c("x", "y"))
facilities <- st_as_sf(data.frame(x = runif(20), y = runif(20)), coords = c("x", "y"))

# Locate 5 facilities minimizing total weighted distance
result <- p_median(demand, facilities, n_facilities = 5, weight_col = "population")

# Mean distance to assigned facility
attr(result, "spopt")$mean_distance
```

route_corridor *Least-Cost Corridor Routing*

Description

Find the minimum-cost path between an origin and destination across a raster friction surface, optionally through an ordered sequence of intermediate waypoints. The cost surface must be in a projected CRS with equal-area cells (not EPSG:4326). Cell values represent traversal friction (cost per unit distance). Higher values = more expensive. NA cells are impassable.

Usage

```
route_corridor(
  cost_surface,
  origin,
  destination,
  waypoints = NULL,
  neighbours = 8L,
  method = c("dijkstra", "bidirectional", "astar"),
  resolution_factor = 1L,
  output = c("combined", "segments")
)
```

Arguments

cost_surface	A terra SpatRaster (single band) or an spopt_corridor_graph object created by corridor_graph. When a graph object is supplied, neighbours and resolution_factor are fixed at graph build time and cannot be overridden.
origin	sf/sfc POINT (single feature) or numeric vector c(x, y) in the CRS of cost_surface. Must fall on a non-NA cell.
destination	sf/sfc POINT (single feature) or numeric vector c(x, y). Must fall on a non-NA cell.
waypoints	Optional intermediate points in the required visit order. NULL (default) for point-to-point routing. Otherwise an sf or sfc POINT collection, or a list of numeric c(x, y) vectors / single-point sf/sfc features. Each waypoint must fall on a non-NA cell. Interior waypoints are represented in the output geometry by their snapped cell center (origin and destination retain their exact user-supplied coordinates) to avoid join artifacts at off-cell-center waypoints.
neighbours	Integer. Cell connectivity: 4, 8 (default), or 16. 4 = cardinal only. 8 = cardinal + diagonal. 16 = adds knight's-move. Cannot be overridden when cost_surface is a corridor graph (fixed at corridor_graph build time).
method	Character. Routing algorithm: • "dijkstra" (default): standard Dijkstra's shortest path. • "bidirectional": bidirectional Dijkstra, ~2x faster. • "astar": A* with Euclidean heuristic, fastest for distant pairs.

`resolution_factor`
: Integer, default 1L. Values > 1 aggregate the cost surface before routing (e.g., 2L halves resolution). Cannot be overridden when `cost_surface` is a corridor graph (fixed at `corridor_graph` build time).

`output`
: Character. Output shape when `waypoints` is non-NULL or `"segments"` is explicitly requested:

- "combined" (default): a single-row sf LINESTRING for the full route, with per-segment breakdown in the `"spopt"` attribute.
- "segments": an N+1-row sf, one LINESTRING per leg (class `spopt_corridor_segments`), with per-leg columns.

Details

Ordered-waypoint routing returns the exact minimum-cost route through the required waypoint sequence under the additive raster-cost model used here – shortest paths compose, so chaining segments between consecutive required points is optimal for the ordered-via formulation. Two related problems are *not* solved by this API and belong to future extensions: choosing the best visit order for unordered waypoints (a TSP-style problem) and connecting multiple required cities with a branched minimum-cost network on the cost surface (a Steiner tree problem).

Value

An sf LINESTRING object with columns:

- `total_cost`: accumulated traversal cost (friction * distance units)
- `n_cells`: number of cells in the path
- `straight_line_dist`: Euclidean distance origin to destination (CRS units)
- `path_dist`: actual path length (CRS units)
- `sinuosity`: path_dist / straight_line_dist (NA if origin == destination)

When waypoints are supplied and `output = "combined"`, two additional columns report the leg-sum baseline: `leg_straight_line_dist` (sum of straight-line distances between consecutive ordered points) and `leg_sinuosity` (`path_dist / leg_straight_line_dist`). `sinuosity` retains its origin-to-destination semantics and will grow with the number of waypoints, as expected.

The returned linestring starts at the user-supplied origin coordinates, passes through the cell centers along the optimal path (including the snapped cell center at each interior waypoint), and ends at the user-supplied destination coordinates.

The "spopt" attribute contains metadata including `total_cost`, `n_cells`, `method`, `neighbours`, `solve_time`, `graph_build_time`, `cell_indices`, and grid dimensions. When any waypoints are supplied, the attribute also contains `n_waypoints_input` (count as originally supplied), `n_waypoints_effective` (count after eliding exact duplicates and same-cell collapses; may be less than `n_waypoints_input`), `n_segments_effective`, `waypoints_input_xy` (`n_waypoints_input`-by-2 matrix of exact user-supplied coords – always preserved, independent of elision), `waypoint_cell_xy` and `waypoint_cells` (length `n_waypoints_effective`, the snapped cells actually routed through), `segment_costs`, `segment_path_dists`, `segment_cells`, and `segment_solve_times`. `cell_indices` in combined mode is the concatenated routed path after shared-join dedup – the cells physically traversed, not a record of supplied waypoints.

With `output = "segments"`, the return is an sf with N+1 rows (one LINESTRING per leg) with columns `segment`, `from_label`, `to_label`, per-leg coordinates, `total_cost`, `n_cells`, `path_dist`, `straight_line_dist`, `sinuosity`. The object-level `"spopt"` attribute carries the same aggregate metadata as combined mode.

Examples

```
library(terra)
library(sf)

# Build a simple friction surface (projected CRS)
r <- rast(nrows = 500, ncols = 500, xmin = 0, xmax = 500000,
          ymin = 0, ymax = 500000, crs = "EPSG:32614")
values(r) <- runif(ncell(r), 0.5, 2.0)

origin <- st_sfc(st_point(c(50000, 50000)), crs = 32614)
dest   <- st_sfc(st_point(c(450000, 450000)), crs = 32614)

path <- route_corridor(r, origin, dest)
plot(r)
plot(st_geometry(path), add = TRUE, col = "red", lwd = 2)

# Ordered multi-city corridor
wp <- list(c(200000, 250000), c(350000, 350000))
via_path <- route_corridor(r, origin, dest, waypoints = wp)
via_segs <- route_corridor(r, origin, dest, waypoints = wp,
                           output = "segments")

# Graph caching for multiple OD pairs or repeated waypoint routes
g <- corridor_graph(r, neighbours = 8L)
path1 <- route_corridor(g, origin, dest, method = "astar")
path2 <- route_corridor(g, origin, dest, waypoints = wp)
```

`route_k_corridors` *k-Diverse Corridor Routing*

Description

Find k spatially distinct corridors between an origin and destination using iterative penalty rerouting. The function produces a ranked set of geographically different route alternatives rather than minor variants of the same optimal path.

Usage

```
route_k_corridors(
  cost_surface,
  origin,
```

```
  destination,
  k = 5L,
  penalty_radius = NULL,
  penalty_factor = 2,
  min_spacing = NULL,
  neighbours = 8L,
  method = c("dijkstra", "bidirectional", "astar"),
  resolution_factor = 1L
)
```

Arguments

cost_surface	A terra SpatRaster (single band). Same requirements as route_corridor. Must NOT be a corridor_graph object (the cost surface is modified each iteration).
origin	sf/sfc POINT or numeric c(x, y). Same as route_corridor.
destination	sf/sfc POINT or numeric c(x, y). Must differ from origin.
k	Integer >= 1. Number of diverse corridors to find. May return fewer if no feasible alternative exists.
penalty_radius	Numeric. Buffer distance (CRS units) around each found path within which cells are penalized. If NULL (default), uses 5 percent of the straight-line distance between origin and destination.
penalty_factor	Numeric > 1.0. Multiplier applied to cells within the penalty radius. Cumulative: a cell near paths 1 and 2 is multiplied by penalty_factor^2. Default 2.0.
min_spacing	Numeric or NULL. Minimum average distance (CRS units) between a candidate and its nearest prior accepted path. Candidates below this threshold are discarded and the iteration retries with increased penalty (up to 3 retries per rank). NULL (default) disables the check.
neighbours	Integer. Cell connectivity: 4, 8 (default), or 16.
method	Character. Routing algorithm: "dijkstra", "bidirectional", or "astar" (default, recommended for iterative use).
resolution_factor	Integer >= 1. Aggregation factor applied once before routing begins.

Details

In corridor planning, the mathematically optimal route is frequently infeasible due to factors not fully captured in the cost surface: landowner opposition, permitting constraints, or environmental findings during field survey. Presenting k diverse alternatives allows planners to evaluate trade-offs and select routes that are robust to on-the-ground uncertainty.

The function uses a **heuristic buffered penalty rerouting** method adapted for raster friction surfaces. After finding the least-cost path, cells within a geographic buffer (penalty_radius) are penalized by multiplying their friction by penalty_factor. The solver then runs again on the modified surface. Each accepted path's penalty is cumulative, pushing subsequent corridors into genuinely different geography.

The general idea of iterative penalty rerouting originates in the alternative route generation literature (de la Barra et al., 1993), where it was applied to road network graphs. Abraham et al. (2013)

evaluate it as one of several alternative route strategies for road networks. This implementation adapts the concept to raster surfaces by penalizing a spatial buffer zone around prior paths (rather than exact reused edges), which directly targets geographic separation at the corridor scale.

This approach is distinct from the formal k-shortest paths with limited overlap (kSPwLO) algorithms of Chondrogiannis et al. (2020), which guarantee overlap-bounded optimality through edge elimination. The penalty method does not guarantee the k cheapest diverse paths; it is a heuristic that produces corridors which are reliably spatially distinct and reasonably close in cost. Talsma et al. (2026) find that alternative pipeline corridors produced by a kSPwLO approach cost only 4.5–6.2 percent more than the optimal route, suggesting that the cost of diversity is modest in practice.

Each corridor's `total_cost` is recomputed on the **original** (unpenalized) surface so that costs are directly comparable across ranks. Diversity metrics (`mean_spacing`, `pct_overlap`) are measured against the rank-1 path to answer the planning question: "how different is this alternative from the route we would build first?"

Value

An sf object with up to k rows, each a LINESTRING, with columns:

alternative Integer. 1 = optimal path on the original surface, 2+ = alternatives generated by iterative penalty rerouting. Alternatives are not guaranteed to increase monotonically in cost.

total_cost Accumulated traversal cost on the **original** (unpenalized) surface, allowing fair comparison across ranks.

n_cells Number of raster cells in the path.

path_dist Path length in CRS units.

straight_line_dist Euclidean distance from origin to destination.

sinuosity path_dist / straight_line_dist.

mean_spacing Mean distance from this path's vertices to the nearest point on the rank-1 path. NA for rank 1.

pct_overlap Fraction of this path's cells within `penalty_radius` of the rank-1 path. NA for rank 1.

Class is `c("spopt_k_corridors", "sf")`. The `"spopt"` attribute contains: `k_requested`, `k_found`, `penalty_radius`, `penalty_factor`, `min_spacing`, `total_solve_time`, `total_graph_build_time`, `total_retries`, `method`, `neighbours`, `resolution_factor`.

References

Chondrogiannis, T., Bouros, P., Gamper, J., Leser, U., & Blumenthal, D. B. (2020). Finding k-Shortest Paths with Limited Overlap. The VLDB Journal, 29(5), 1023–1047. doi:10.1007/s00778-02000604x

Abraham, I., Delling, D., Goldberg, A. V., & Werneck, R. F. (2013). Alternative Routes in Road Networks. ACM Journal of Experimental Algorithmics, 18(1), 1–17. doi:10.1145/2444016.2444019

de la Barra, T., Perez, B., & Anez, J. (1993). Multidimensional Path Search and Assignment. Proceedings of the 21st PTRC Summer Annual Meeting, Manchester, UK.

Talsma, C. J., Duque, J. C., Prehn, J., Upchurch, C., Lim, S., & Middleton, R. S. (2026). Identifying Critical Pathways in CO2 Pipeline Routing using a K-shortest Paths with Limited Overlap Algorithm. Preprint. doi:10.31224/6670

See Also

route_corridor() for single least-cost paths, corridor_graph() for cached graph routing with multiple OD pairs

Examples

```
library(terra); library(sf)
r <- rast(nrows = 200, ncols = 200, xmin = 0, xmax = 200000,
          ymin = 0, ymax = 200000, crs = "EPSG:32614")
values(r) <- runif(ncell(r), 0.5, 2.0)

# Find 5 diverse alternatives
result <- route_k_corridors(r, c(10000, 10000), c(190000, 190000), k = 5)
print(result)
plot(result)

# Enforce minimum geographic separation between alternatives
result <- route_k_corridors(r, c(10000, 10000), c(190000, 190000),
                            k = 5, min_spacing = 10000)
```

---

route_tsp *Traveling Salesman Problem (TSP)*

Description

Solves fixed-start routing problems over an sf layer. By default the route is closed (start and end at the same location), but open routes and fixed start/end paths are also supported. Optional time windows can be supplied in the same units as the travel-time matrix.

Usage

```
route_tsp(
  locations,
  depot = 1L,
  start = NULL,
  end = NULL,
  cost_matrix = NULL,
  distance_metric = "euclidean",
  method = "2-opt",
  earliest = NULL,
  latest = NULL,
  service_time = NULL
)
```

Arguments

locations	An sf object representing locations to visit.
depot	Integer. Backward-compatible alias for start when start is not supplied. Defaults to 1.
start	Integer. Row index of the route start in locations (1-based). If omitted, depot is used.
end	Integer or NULL. Row index of the route end in locations (1-based). If omitted, defaults to start for a closed route. Set to NULL explicitly for an open route that may end at any stop.
cost_matrix	Optional. Pre-computed square distance/cost matrix (n x n). If NULL, computed from locations using distance_metric.
distance_metric	Distance metric when computing from geometry: "euclidean" (default) or "manhattan".
method	Algorithm: "2-opt" (default, nearest-neighbor + local search) or "nn" (nearest-neighbor only).
earliest	Optional earliest arrival/service times. Supply either a numeric vector of length nrow(locations) or a column name in locations.
latest	Optional latest arrival/service times. Must be supplied together with earliest.
service_time	Optional service duration at each stop. Supply either a numeric vector of length nrow(locations) or a column name in locations.

Details

Supported route variants:

1. **Closed tour**: `start = 1, end = 1` (default)
2. **Open route**: `start = 1, end = NULL`
3. **Fixed path**: `start = 1, end = 5`

Time windows use the same units as cost_matrix. When windows are supplied, the solver constructs a feasible route and only accepts local-search moves that preserve feasibility.

Value

An sf object (the input locations) with added columns:

- .visit_order: Visit sequence (1 = route start)
- .tour_position: Position in the returned tour/path
- .arrival_time: Arrival/service start time at each visited stop
- .departure_time: Departure time after service

Metadata is stored in the "spopt" attribute, including total_cost, nn_cost, improvement_pct, tour, start, end, route_type, and solve_time.

See Also

route_vrp() for multi-vehicle routing, distance_matrix() for computing cost matrices

route_vrp *Vehicle Routing Problem (VRP)*

Description

Solves the Capacitated Vehicle Routing Problem (CVRP): find minimum-cost routes for a fleet of vehicles, each with a capacity limit, to serve all customers from a central depot. Uses Clarke-Wright savings heuristic for construction with 2-opt, relocate, and swap local search improvement.

Usage

```
route_vrp(
  locations,
  depot = 1L,
  demand_col,
  vehicle_capacity,
  n_vehicles = NULL,
  cost_matrix = NULL,
  distance_metric = "euclidean",
  method = "2-opt",
  service_time = NULL,
  max_route_time = NULL,
  balance = NULL,
  earliest = NULL,
  latest = NULL
)
```

Arguments

locations	An sf object representing all locations (depot + customers).
depot	Integer. Row index of the depot in locations (1-based). Defaults to 1.
demand_col	Character. Column name in locations containing the demand at each stop. The depot's demand is ignored (set to 0 internally).
vehicle_capacity	Numeric. Maximum capacity per vehicle.
n_vehicles	Integer or NULL. Maximum number of vehicles. If NULL (default), uses as many as needed to satisfy capacity constraints.
cost_matrix	Optional. Pre-computed square cost/distance matrix (n x n). If NULL, computed from locations using distance_metric.
distance_metric	Distance metric when computing from geometry: "euclidean" (default) or "manhattan".
method	Algorithm: "2-opt" (default, savings + local search) or "savings" (Clarke-Wright construction only).

service_time	Optional service/dwell time at each stop. Supply either a numeric vector of length nrow(locations) or a column name in locations. The depot's service time is forced to 0.
max_route_time	Optional maximum total time per route (travel time + service time). Routes that would exceed this limit are split.
balance	Optional balancing mode. Currently supports "time" to minimize the longest route's total time after cost optimization. This runs a post-optimization phase that may slightly increase total cost (bounded to a small percentage). Ignored when method = "savings".
earliest	Optional earliest arrival/service time at each stop. Supply either a numeric vector of length nrow(locations) or a column name in locations. Must be supplied together with latest.
latest	Optional latest arrival/service time at each stop. Must be supplied together with earliest. A vehicle may arrive early and wait, but cannot begin service after this time.

Details

The CVRP extends the TSP to multiple vehicles with capacity constraints. The solver uses a two-phase approach:

1. **Construction** (Clarke-Wright savings): Starts with one route per customer, then iteratively merges routes that produce the greatest distance savings while respecting capacity limits.
2. **Improvement** (if method = "2-opt"): Applies intra-route 2-opt (reversing subtours), inter-route relocate (moving a customer between routes), and inter-route swap (exchanging customers between routes).

Value

An sf object (the input locations) with added columns:

- .vehicle: Vehicle assignment (1, 2, ...). Depot is 0.
- .visit_order: Visit sequence within each vehicle's route.

Metadata is stored in the "spopt" attribute, including n_vehicles, total_cost, total_time, per-vehicle vehicle_costs, vehicle_times, vehicle_loads, and vehicle_stops.

Units and cost matrices

The cost_matrix can contain travel times, distances, or any generalized cost. The solver minimizes the sum of matrix values along each route. When using time-based features (service_time, max_route_time, earliest/latest, balance = "time"), the cost matrix should be in time-compatible units (e.g., minutes). Otherwise the time-based outputs and constraints will be dimensionally inconsistent.

In the per-vehicle summary, **Cost** is the raw matrix objective (travel only), while **Time** adds service duration and any waiting induced by time windows. When no service time or windows are used, Time equals Cost and is not shown separately.

Use Cases

- **Oilfield logistics**: Route vacuum trucks to well pads with fluid volume constraints
- **Delivery routing**: Multiple delivery vehicles with weight/volume limits
- **Service dispatch**: Assign and sequence jobs across a fleet of technicians or crews
- **Waste collection**: Route collection vehicles with load capacity

See Also

`route_tsp()` for single-vehicle routing

Examples

```
library(sf)

# Depot + 20 customers with demands
locations <- st_as_sf(
  data.frame(
    id = 1:21, x = runif(21), y = runif(21),
    demand = c(0, rpois(20, 10))
  ),
  coords = c("x", "y")
)

result <- route_vrp(locations, depot = 1, demand_col = "demand", vehicle_capacity = 40)

# How many vehicles needed?
attr(result, "spopt")$n_vehicles

# Per-vehicle costs
attr(result, "spopt")$vehicle_costs
```

---

rust_azp *Solve AZP regionalization problem*

Description

Automatic Zoning Procedure with basic, tabu, and SA variants.

Usage

```
rust_azp(
  attrs,
  n_regions,
  adj_i,
  adj_j,
  method,
```

```
  max_iterations,
  tabu_length,
  cooling_rate,
  initial_temperature,
  seed
)
```

Arguments

attrs	Attribute matrix (n x p)
n_regions	Number of regions to create
adj_i	Row indices of adjacency (0-based)
adj_j	Column indices of adjacency (0-based)
method	"basic", "tabu", or "sa"
max_iterations	Maximum iterations
tabu_length	Tabu list length (for tabu method)
cooling_rate	SA cooling rate (for sa method)
initial_temperature	SA initial temperature (for sa method)
seed	Random seed

Value

List with labels, n_regions, objective

---

rust_connected_components

Find connected components

Description

Find connected components

Usage

```
rust_connected_components(i, j, n)
```

Arguments

i	Row indices of adjacency matrix
j	Column indices of adjacency matrix
n	Number of nodes

Value

Vector of component labels (0-based)

`rust_corridor` *Least-cost corridor routing on a raster grid*

Description

Least-cost corridor routing on a raster grid

Usage

```
rust_corridor(
  values,
  n_rows,
  n_cols,
  cell_width,
  cell_height,
  origin_cell,
  dest_cell,
  neighbours,
  method
)
```

Arguments

`values`	Flattened cell values (row-major), NaN = impassable
`n_rows`	Number of rows in the raster
`n_cols`	Number of columns in the raster
`cell_width`	Cell width in CRS units
`cell_height`	Cell height in CRS units
`origin_cell`	0-based cell index of origin
`dest_cell`	0-based cell index of destination
`neighbours`	Cell connectivity: 4, 8, or 16
`method`	Routing algorithm: "dijkstra", "bidirectional", or "astar"

Value

List with path_cells, total_cost, solve_time_ms, graph_build_time_ms, n_edges

`rust_distance_matrix_euclidean` *Compute Euclidean distance matrix between two sets of points*

Description

Compute Euclidean distance matrix between two sets of points

Usage

```
rust_distance_matrix_euclidean(x1, y1, x2, y2)
```

Arguments

`x1`	X coordinates of first set of points
`y1`	Y coordinates of first set of points
`x2`	X coordinates of second set of points
`y2`	Y coordinates of second set of points

Value

Distance matrix (n1 x n2)

`rust_distance_matrix_manhattan` *Compute Manhattan distance matrix between two sets of points*

Description

Compute Manhattan distance matrix between two sets of points

Usage

```
rust_distance_matrix_manhattan(x1, y1, x2, y2)
```

Arguments

`x1`	X coordinates of first set of points
`y1`	Y coordinates of first set of points
`x2`	X coordinates of second set of points
`y2`	Y coordinates of second set of points

Value

Distance matrix (n1 x n2)

rust_frlm_greedy *Solve FRLM using greedy heuristic*

Description

Solve FRLM using greedy heuristic

Usage

```
rust_frlm_greedy(
  n_candidates,
  path_candidates,
  path_offsets,
  path_distances,
  flow_volumes,
  vehicle_range,
  n_facilities
)
```

Arguments

n_candidates	Number of candidate facility locations
path_candidates	Flat array of candidate indices for each path
path_offsets	Start index for each path in path_candidates
path_distances	Distances to each candidate along paths
flow_volumes	Volume of each flow
vehicle_range	Maximum vehicle range
n_facilities	Number of facilities to place

Value

List with selected facilities and coverage info

rust_huff *Compute Huff Model probabilities*

Description

Computes probability surface based on distance decay and attractiveness. Formula: $P_{ij} = (A_j \times D_{ij}^{\beta}) / \Sigma_k (A_k \times D_{ik}^{\beta})$

Usage

```
rust_huff(cost_matrix, attractiveness, distance_exponent, sales_potential)
```

Arguments

cost_matrix	Cost/distance matrix (demand x stores)
attractiveness	Attractiveness values for each store (pre-computed with exponents)
distance_exponent	Distance decay exponent (typically negative, e.g., -1.5)
sales_potential	Optional sales potential for each demand point

Value

List with probabilities, market shares, expected sales

---

rust_is_connected *Check if a graph is connected*

Description

Check if a graph is connected

Usage

```
rust_is_connected(i, j, n)
```

Arguments

i	Row indices of adjacency matrix
j	Column indices of adjacency matrix
n	Number of nodes

Value

TRUE if connected, FALSE otherwise

rust_mst *Compute minimum spanning tree from adjacency matrix*

Description

Compute minimum spanning tree from adjacency matrix

Usage

```
rust_mst(i, j, weights, n)
```

Arguments

i	Row indices (0-based) of adjacency matrix non-zero entries
j	Column indices (0-based) of adjacency matrix non-zero entries
weights	Edge weights (distances/dissimilarities)
n	Number of nodes

Value

List with MST edges (from, to, weight)

rust_skater *Solve SKATER regionalization*

Description

Solve SKATER regionalization

Usage

```
rust_skater(attrs, adj_i, adj_j, n_regions, floor_var, floor_value, seed)
```

Arguments

attrs	Attribute matrix (n x p)
adj_i	Row indices of adjacency
adj_j	Column indices of adjacency
n_regions	Number of regions to create
floor_var	Optional floor variable values
floor_value	Minimum floor value per region
seed	Random seed

Value

Vector of region labels (1-based)

rust_spenc *Solve SPENC regionalization problem*

Description

Spatially-encouraged spectral clustering.

Usage

```
rust_spenc(attrs, n_regions, adj_i, adj_j, gamma, seed)
```

Arguments

attrs	Attribute matrix (n x p)
n_regions	Number of regions to create
adj_i	Row indices of adjacency (0-based)
adj_j	Column indices of adjacency (0-based)
gamma	RBF kernel parameter
seed	Random seed

Value

List with labels, n_regions, objective

rust_tsp *Solve Traveling Salesman Problem (TSP)*

Description

Solve a closed tour, open route, or fixed-end path over a square cost/distance matrix using nearest-neighbor construction with optional 2-opt and or-opt local search. Optional time windows and service times can be supplied for each stop.

Usage

```
rust_tsp(cost_matrix, start, end, method, earliest, latest, service_time)
```

Arguments

cost_matrix	Square cost/distance matrix (n x n)
start	Start index (0-based)
end	End index (0-based), or NULL for an open route
method	Algorithm: "nn" (nearest-neighbor only) or "2-opt" (with local search)
earliest	Optional earliest service times
latest	Optional latest service times
service_time	Optional service times at each stop

Value

List with tour (1-based), total_cost, nn_cost, improvement_pct, iterations, arrival_time, and departure_time

---

`rust_vrp` *Solve Capacitated Vehicle Routing Problem (CVRP)*

Description

Find minimum-cost routes for multiple vehicles, each with a capacity limit, to serve all customers from a depot. Uses Clarke-Wright savings heuristic with 2-opt, relocate, and swap improvement.

Usage

```
rust_vrp(
  cost_matrix,
  depot,
  demands,
  capacity,
  max_vehicles,
  method,
  service_times,
  max_route_time,
  balance_time,
  earliest,
  latest
)
```

Arguments

`cost_matrix`	Square cost/distance matrix (n x n)
`depot`	Depot index (0-based)
`demands`	Demand at each location (depot demand should be 0)
`capacity`	Vehicle capacity
`max_vehicles`	Maximum number of vehicles (NULL for unlimited)
`method`	Algorithm: "savings" (construction only) or "2-opt" (with improvement)
`service_times`	Optional service time at each stop (NULL for zero)
`max_route_time`	Optional maximum total time per route (NULL for unlimited)
`balance_time`	Whether to run route-time balancing phase
`earliest`	Optional earliest arrival times at each stop
`latest`	Optional latest arrival times at each stop

Value

List with vehicle assignments, visit orders, costs, and route details

rust_ward_constrained *Solve spatially-constrained Ward clustering*

Description

Solve spatially-constrained Ward clustering

Usage

```
rust_ward_constrained(attrs, n_regions, adj_i, adj_j)
```

Arguments

attrs	Attribute matrix (n x p)
n_regions	Number of regions to create
adj_i	Row indices of adjacency (0-based)
adj_j	Column indices of adjacency (0-based)

Value

List with labels, n_regions, objective

second_areal_moment *Second areal moment (i.e., second moment of inertia)*

Description

Computes the second moment of area (also known as the second moment of inertia) for polygon geometries. This is a measure of how the area of a shape is distributed relative to its centroid.

Usage

```
second_areal_moment(x, project = TRUE)
```

Arguments

x	An sf object, sfc geometry column, or sfg geometry.
project	Logical. If the geometries have geodetic coordinates, then they will be projected using an Albers Equal Area Conic projection centered on the data.

Details

The second moment of area is the sum of the inertia across the x and y axes:

The inertia for the x axis is:

$$I_x = \frac{1}{12}\sum_{i=1}^{N}(x_i y_{i+1} - x_{i+1} y_i)(x_i^2 + x_i x_{i+1} + x_{i+1}^2)$$

While the y axis is in a similar form:

$$I_y = \frac{1}{12}\sum_{i=1}^{N}(x_i y_{i+1} - x_{i+1} y_i)(y_i^2 + y_i y_{i+1} + y_{i+1}^2)$$

where x_i, y_i is the current point and x_{i+1}, y_{i+1} is the next point, and where $x_{n+1} = x_1, y_{n+1} = y_1$.

For multipart polygons with holes, all parts are treated as separate contributions to the overall centroid, which provides the same result as if all parts with holes are separately computed, and then merged together using the parallel axis theorem.

The code and documentation are adapted from the PySAL Python package (Ray and Anselin, 2007). See Hally (1987) and Li et al. (2013) for additional details.

Value

Numeric vector of second areal moments

References

Hally, D. 1987. "The calculations of the moments of polygons." Canadian National Defense Research and Development Technical Memorandum 87/209. https://apps.dtic.mil/sti/tr/pdf/ADA183444.pdf

Li, W., Goodchild, M.F., and Church, R.L. 2013. "An Efficient Measure of Compactness for Two-Dimensional Shapes and Its Application in Regionalization Problems." International Journal of Geographical Information Science 27 (6): 1227–50. doi:10.1080/13658816.2012.752093.

Rey, Sergio J., and Luc Anselin. 2007. "PySAL: A Python Library of Spatial Analytical Methods." Review of Regional Studies 37 (1): 5–27. doi:10.52324/001c.8285.

See Also

`nmi()`, which computes the normalized moment of inertia.

Examples

```
library(sf)
poly <- st_polygon(list(matrix(c(0,0, 1,0, 1,1, 0,1, 0,0), ncol=2, byrow=TRUE)))
second_areal_moment(poly)
```

skater *SKATER Spatial Clustering*

Description

Performs spatial clustering using the SKATER algorithm (Spatial 'K'luster Analysis by Tree Edge Removal). The algorithm builds a minimum spanning tree from the spatial contiguity graph, then iteratively removes edges to create spatially contiguous clusters.

Usage

```
skater(
  data,
  attrs = NULL,
  n_regions,
  weights = "queen",
  bridge_islands = FALSE,
  floor = NULL,
  floor_value = 0,
  scale = TRUE,
  seed = NULL,
  verbose = FALSE
)
```

Arguments

`data`	An sf object with polygon or point geometries.
`attrs`	Character vector of column names to use for clustering (e.g., `c("var1", "var2")`). If NULL, uses all numeric columns.
`n_regions`	Integer. Number of regions (clusters) to create.
`weights`	Spatial weights specification. Can be: • `"queen"` (default): Polygons sharing any boundary point are neighbors • `"rook"`: Polygons sharing an edge are neighbors • An nb object from spdep or created with `sp_weights()` • A list for other weight types: `list(type = "knn", k = 6)` for k-nearest neighbors, or `list(type = "distance", d = 5000)` for distance-based weights
`bridge_islands`	Logical. If TRUE, automatically connect disconnected components (e.g., islands) using nearest-neighbor edges. If FALSE (default), the function will error when the spatial weights graph is disconnected.
`floor`	Optional. Column name specifying a floor constraint variable.
`floor_value`	Numeric. Minimum sum of floor variable required per region. Only used if `floor` is specified.
`scale`	Logical. If TRUE (default), standardize attributes before clustering.
`seed`	Optional integer for reproducibility.
`verbose`	Logical. Print progress messages.

Value

An sf object with a `.region` column containing cluster assignments. Metadata is stored in the "spopt" attribute.

References

Assuncao, R. M., Neves, M. C., Camara, G., & Freitas, C. da C. (2006). Efficient regionalization techniques for socio-economic geographical units using minimum spanning trees. International Journal of Geographical Information Science, 20(7), 797-811.

Examples

```
library(sf)
nc <- st_read(system.file("shape/nc.shp", package = "sf"))

# Cluster into 5 regions based on SIDS rates
result <- skater(nc, attrs = c("SID74", "SID79"), n_regions = 5)

# With floor constraint: each region must have at least 100,000 births
result <- skater(nc, attrs = c("SID74", "SID79"), n_regions = 5,
                 floor = "BIR74", floor_value = 100000)

# View results
plot(result[".region"])
```

---

spenc *Spatially-Encouraged Spectral Clustering (SPENC)*

Description

Performs spectral clustering with spatial constraints by combining spatial connectivity with attribute similarity using kernel methods. This approach is useful for clustering with highly non-convex clusters or irregular topologies in geographic contexts.

Usage

```
spenc(
  data,
  attrs = NULL,
  n_regions,
  weights = "queen",
  bridge_islands = FALSE,
  gamma = 1,
  scale = TRUE,
  seed = NULL,
  verbose = FALSE
)
```

Arguments

`data`	An sf object with polygon or point geometries.
`attrs`	Character vector of column names to use for clustering (e.g., `c("var1", "var2")`). If NULL, uses all numeric columns.
`n_regions`	Integer. Number of regions (clusters) to create.
`weights`	Spatial weights specification. Can be: • `"queen"` (default): Polygons sharing any boundary point are neighbors • `"rook"`: Polygons sharing an edge are neighbors • An `nb` object from spdep or created with `sp_weights()` • A list for other weight types: `list(type = "knn", k = 6)` for k-nearest neighbors, or `list(type = "distance", d = 5000)` for distance-based weights
`bridge_islands`	Logical. If TRUE, automatically connect disconnected components (e.g., islands) using nearest-neighbor edges. If FALSE (default), the function will error when the spatial weights graph is disconnected.
`gamma`	Numeric. RBF kernel parameter controlling attribute similarity decay. Larger values = faster decay = more local similarity. Default is 1. Can also be "auto" to estimate from data.
`scale`	Logical. If TRUE (default), standardize attributes before clustering.
`seed`	Optional integer for reproducibility.
`verbose`	Logical. Print progress messages.

Details

SPENC (Wolf, 2021) extends spectral clustering to incorporate spatial constraints. The algorithm:

1. Computes attribute affinity using an RBF (Gaussian) kernel
2. Multiplies element-wise with spatial weights (only neighbors have affinity)
3. Computes the normalized Laplacian of the combined affinity matrix
4. Extracts the k smallest eigenvectors as a spectral embedding
5. Applies k-means clustering to the embedding

Key advantages:

- Can find non-convex cluster shapes
- Respects spatial connectivity
- Balances attribute similarity with spatial proximity

The gamma parameter controls how quickly attribute similarity decays with distance in attribute space. Larger values create more localized clusters.

Value

An sf object with a `.region` column containing cluster assignments. Metadata is stored in the "spopt" attribute, including:

- algorithm: "spenc"
- n_regions: Number of regions created
- objective: Within-cluster sum of squared distances in embedding space
- gamma: The gamma parameter used
- solve_time: Time to solve in seconds

References

Wolf, L. J. (2021). Spatially-encouraged spectral clustering: a technique for blending map typologies and regionalization. International Journal of Geographical Information Science, 35(11), 2356-2373. doi:10.1080/13658816.2021.1934475

Examples

```
library(sf)
nc <- st_read(system.file("shape/nc.shp", package = "sf"))

# Basic SPENC with 8 regions
result <- spenc(nc, attrs = c("SID74", "SID79"), n_regions = 8)

# Adjust gamma for different cluster tightness
result <- spenc(nc, attrs = c("SID74", "SID79"), n_regions = 8, gamma = 0.5)

# View results
plot(result[".region"])
```

---

sp_weights *Create spatial weights from an sf object*

Description

Constructs spatial weights (neighborhood structure) from sf geometries. Wraps spdep functions with a convenient interface.

Usage

```
sp_weights(
  data,
  type = c("queen", "rook", "knn", "distance"),
  k = NULL,
  d = NULL,
  ...
)
```

Arguments

data	An sf object with polygon or point geometries.
type	Type of weights. One of: • "queen" (default): Polygons sharing any boundary point are neighbors • "rook": Polygons sharing an edge are neighbors • "knn": K-nearest neighbors based on centroid distance • "distance": All units within a distance threshold are neighbors
k	Number of nearest neighbors. Required when type = "knn".
d	Distance threshold. Required when type = "distance". Units match the CRS of the data (e.g., meters for projected CRS).
...	Additional arguments passed to spdep functions.

Details

Choosing a weight type:

- Use **queen/rook** for polygon data where physical adjacency matters
- Use **knn** when you need guaranteed connectivity (no isolates) or for point data
- Use **distance** for point data or when interaction depends on proximity

KNN weights always produce a connected graph (if k >= 1), making them useful for datasets with islands or disconnected polygons.

Value

A neighbors list object (class "nb") compatible with spdep.

Examples

```
library(sf)
nc <- st_read(system.file("shape/nc.shp", package = "sf"))

# Queen contiguity (default)
w_queen <- sp_weights(nc, type = "queen")

# K-nearest neighbors (guarantees connectivity)
w_knn <- sp_weights(nc, type = "knn", k = 6)

# Distance-based (e.g., 50km for projected data)
nc_proj <- st_transform(nc, 32119)  # NC State Plane
w_dist <- sp_weights(nc_proj, type = "distance", d = 50000)
```

`tarrant_travel_times` *Tarrant County Travel Time Matrix Example Data*

Description

A dataset containing Census tract polygons, candidate facility points, and a pre-computed driving travel-time matrix for Tarrant County, Texas. Used for demonstrating facility location algorithms with real travel times.

Usage

```
tarrant_travel_times
```

Format

A list with four elements:

tracts An sf object with Census tract polygons including GEOID, NAME, population, and geometry columns.

demand An sf object with tract centroid points (demand locations) in WGS84 coordinates.

candidates An sf object with 30 randomly sampled candidate facility locations in WGS84 coordinates.

matrix A numeric matrix of driving travel times in minutes. Rows correspond to demand points, columns to candidate facilities. Unreachable pairs are set to Inf.

Details

The travel-time matrix was generated using r5r with OpenStreetMap road network data clipped to Tarrant County. Candidate facilities were sampled using `set.seed(1983)` for reproducibility.

Source

Census tract data from the American Community Survey via tidycensus. Road network from OpenStreetMap via GeoFabrik. Travel times computed with r5r.

Examples

```
data(tarrant_travel_times)

# Access components
tracts <- tarrant_travel_times$tracts
demand <- tarrant_travel_times$demand
candidates <- tarrant_travel_times$candidates
ttm <- tarrant_travel_times$matrix

# Use with p_median
result <- p_median(
  demand = demand,
```

```
  facilities = candidates,
  n_facilities = 5,
  weight_col = "population",
  cost_matrix = ttm
)
```

`ward_spatial` *Ward Spatial Clustering*

Description

Performs spatially-constrained hierarchical clustering using Ward's minimum variance method. Only spatially contiguous areas can be merged, ensuring all resulting regions are spatially connected.

Usage

```
ward_spatial(
  data,
  attrs = NULL,
  n_regions,
  weights = "queen",
  bridge_islands = FALSE,
  scale = TRUE,
  verbose = FALSE
)
```

Arguments

`data`	An sf object with polygon or point geometries.
`attrs`	Character vector of column names to use for clustering (e.g., `c("var1", "var2")`). If NULL, uses all numeric columns.
`n_regions`	Integer. Number of regions (clusters) to create.
`weights`	Spatial weights specification. Can be: • `"queen"` (default): Polygons sharing any boundary point are neighbors • `"rook"`: Polygons sharing an edge are neighbors • An nb object from spdep or created with `sp_weights()` • A list for other weight types: `list(type = "knn", k = 6)` for k-nearest neighbors, or `list(type = "distance", d = 5000)` for distance-based weights
`bridge_islands`	Logical. If TRUE, automatically connect disconnected components (e.g., islands) using nearest-neighbor edges. If FALSE (default), the function will error when the spatial weights graph is disconnected.
`scale`	Logical. If TRUE (default), standardize attributes before clustering.
`verbose`	Logical. Print progress messages.

Details

This function implements spatially-constrained agglomerative hierarchical clustering using Ward's minimum variance criterion. Unlike standard Ward clustering, this version enforces spatial contiguity by only allowing clusters that share a border to be merged.

The algorithm:

1. Starts with each observation as its own cluster
2. At each step, finds the pair of **adjacent** clusters with minimum Ward distance (increase in total within-cluster variance)
3. Merges them into a single cluster
4. Repeats until the desired number of regions is reached

The result guarantees that all regions are spatially contiguous.

Value

An sf object with a `.region` column containing cluster assignments. Metadata is stored in the "spopt" attribute.

Examples

```
library(sf)
nc <- st_read(system.file("shape/nc.shp", package = "sf"))

# Cluster into 8 spatially-contiguous regions
result <- ward_spatial(nc, attrs = c("SID74", "SID79"), n_regions = 8)
plot(result[".region"])
```

Index