# Package 'walkability'

November 17, 2026

**Title** Urban Walkability and Active Transit Network Analysis

**Version** 0.1.0

**Description** High-performance geospatial graph analysis for urban street networks, walkability, and active mobility. Features topological diagnostics and controlled repair for user-provided networks, street-network acquisition from 'OpenStreetMap', multi-threaded Rust kernels for isochrone-based reachable-network length metrics with partial-edge credits, and publication-ready 'ggplot2' visualizations. The accessibility measures draw on Hansen (1959) <doi:10.1080/01944365908978307>; street-network analysis follows Boeing (2017) <doi:10.1016/j.compenvurbsys.2017.05.004>; and betweenness centrality follows Freeman (1977) <doi:10.2307/3033543>.

**License** MIT + file LICENSE

**Copyright** See inst/COPYRIGHTS for copyright and license notices for vendored Rust code.

**URL** https://paulesantos.github.io/walkability/, https://github.com/PaulESantos/walkability

**BugReports** https://github.com/PaulESantos/walkability/issues

**Encoding** UTF-8

**LazyData** true

**SystemRequirements** Cargo (Rust's package manager) (>= 1.80), rustc (>= 1.80)

**Depends** R (>= 4.2.0)

**Imports** cli, ggplot2, rlang, sf, tibble

**Suggests** arrow, bslib, dplyr, igraph, knitr, mapgl, osmdata, patchwork, rmarkdown, scales, sfnetworks, shiny, testthat (>= 3.0.0), tidygraph

**VignetteBuilder** knitr

**Config/rextendr/version** 0.5.0

**Config/roxygen2/version** 8.1.1

**NeedsCompilation** yes

**Author** Paul E. Santos Andrade [aut, cre, cph] (ORCID:
<https://orcid.org/0000-0002-6635-0375>)

**Maintainer** Paul E. Santos Andrade <paulefrens@gmail.com>

**Repository** CRAN

**Date/Publication** 2026-11-17 11:50:02 UTC

# Contents

as_sfnetwork . . . 3
as_tidygraph . . . 4
plot.walk_access . . . 4
plot.walk_betweenness . . . 5
plot.walk_comparison . . . 5
plot.walk_diagnosis . . . 6
plot.walk_isochrone . . . 6
plot.walk_morphology . . . 7
plot.walk_path . . . 7
plot.walk_reach . . . 8
print.walk_access . . . 8
print.walk_betweenness . . . 9
print.walk_comparison . . . 9
print.walk_diagnosis . . . 10
print.walk_isochrone . . . 10
print.walk_morphology . . . 11
print.walk_network . . . 11
print.walk_path . . . 12
print.walk_reach . . . 12
print.walk_scenario_betweenness . . . 13
rust_build_csr . . . 13
rust_compute_access_multi . . . 14
rust_compute_circuity_sample . . . 15
rust_compute_reachable_length . . . 16
rust_isochrone_subgraph . . . 17
rust_shortest_path . . . 17
streets_arequipa . . . 18
streets_cusco . . . 19
streets_lima . . . 19
theme_walkability . . . 20
walk_app . . . 21
walk_betweenness . . . 22
walk_build_network . . . 24
walk_compare_scenarios . . . 25
walk_compute_access . . . 26
walk_compute_circuity . . . 27
walk_compute_reach . . . 28
walk_diagnose_network . . . 29
walk_download_osm . . . 31

walk_intersection_density . . . . . 31
walk_isochrone . . . . . 32
walk_plot_diagnosis . . . . . 33
walk_plot_reach . . . . . 34
walk_scenario_betweenness . . . . . 35
walk_scenario_close . . . . . 36
walk_shortest_path . . . . . 37

**Index** **38**

---

`as_sfnetwork` *Convert Walk Network to sfnetworks Graph*

---

## Description

Converts a `walk_network` object into an `sfnetwork` from the `sfnetworks` package for seamless integration with spatial graph routing and tidygraph analysis workflows.

## Usage

```
as_sfnetwork(x, ...)

## S3 method for class 'walk_network'
as_sfnetwork(x, directed = FALSE, ...)
```

## Arguments

| | |
|---|---|
| `x` | A `walk_network` object. |
| `...` | Additional arguments passed to methods. |
| `directed` | Logical. If `TRUE`, creates a directed graph. Default is `FALSE` (undirected pedestrian network). |

## Value

An `sfnetwork` object if `sfnetworks` is installed.

## Examples

```
data(streets_cusco)
net <- walk_build_network(streets_cusco, verbose = FALSE)
if (requireNamespace("sfnetworks", quietly = TRUE)) {
  sfn <- as_sfnetwork(net)
  print(sfn)
}
```

## as_tidygraph *Convert Walk Network to tidygraph tbl_graph*

### Description

Converts a walk_network object into a tbl_graph from the tidygraph package.

### Usage

```
as_tidygraph(x, ...)

## S3 method for class 'walk_network'
as_tidygraph(x, directed = FALSE, ...)
```

### Arguments

| | |
|---|---|
| x | A walk_network object. |
| ... | Additional arguments passed to methods. |
| directed | Logical. If TRUE, creates a directed graph. Default is FALSE. |

### Value

A tbl_graph object if tidygraph is installed.

### Examples

```
data(streets_cusco)
net <- walk_build_network(streets_cusco, verbose = FALSE)
if (requireNamespace("tidygraph", quietly = TRUE)) {
  tg <- as_tidygraph(net)
  print(tg)
}
```

## plot.walk_access *Plot Method for Walk Access*

### Description

Visualizes the spatial distribution of accessible facilities from each intersection.

### Usage

```
## S3 method for class 'walk_access'
plot(x, category = NULL, ...)
```

### Arguments

| | |
|---|---|
| x | A walk_access object. |
| category | Character string of the category to display (if multiple exist). |
| ... | Additional arguments. |

### Value

A ggplot object.

---

## plot.walk_betweenness *Plot Method for Walk Betweenness*

---

### Description

Plot Method for Walk Betweenness

### Usage

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

### Arguments

| | |
|---|---|
| x | A walk_betweenness object. |
| ... | Additional arguments. |

### Value

A ggplot object.

---

## plot.walk_comparison *Plot Method for Walk Comparison*

---

### Description

Visualizes the closed streets in red alongside the difference in reachable pedestrian length.

### Usage

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

### Arguments

| | |
|---|---|
| x | A walk_comparison object. |
| ... | Additional arguments. |

### Value

A ggplot object.

---

## plot.walk_diagnosis *Plot Method for Street Network Diagnosis*

---

### Description

Visualizes disconnected network components and potential undershoots.

### Usage

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

### Arguments

| | |
|---|---|
| x | A walk_diagnosis object. |
| ... | Additional arguments passed to ggplot2. |

### Value

A ggplot object.

---

## plot.walk_isochrone *Plot Method for Walk Isochrone*

---

### Description

Visualizes multi-threshold isochrone polygons with layered fills.

### Usage

```
## S3 method for class 'walk_isochrone'
plot(x, show_network = TRUE, ...)
```

### Arguments

| | |
|---|---|
| x | A walk_isochrone object. |
| show_network | Logical. If TRUE, plots background street network. |
| ... | Additional arguments. |

### Value

A ggplot object.

## plot.walk_morphology *Plot Method for Walk Morphology*

### Description

Visualizes the spatial layout of cul-de-sacs, T-junctions, and crossroads.

### Usage

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

### Arguments

| | |
|---|---|
| x | A walk_morphology object. |
| ... | Additional arguments. |

### Value

A ggplot object.

## plot.walk_path *Plot Method for Walk Path*

### Description

Visualizes the shortest pedestrian path overlaid on the local street network.

### Usage

```
## S3 method for class 'walk_path'
plot(x, network = NULL, ...)
```

### Arguments

| | |
|---|---|
| x | A walk_path object. |
| network | Optional background walk_network. If NULL, uses the embedded network. |
| ... | Additional arguments. |

### Value

A ggplot object.

## plot.walk_reach *Plot Method for Walk Reach Object*

### Description

Plot Method for Walk Reach Object

### Usage

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

### Arguments

| | |
|---|---|
| x | A walk_reach object. |
| ... | Additional arguments passed to walk_plot_reach(). |

### Value

A ggplot object.

## print.walk_access *Print Method for Walk Access*

### Description

Print Method for Walk Access

### Usage

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

### Arguments

| | |
|---|---|
| x | A walk_access object. |
| ... | Additional arguments. |

### Value

The walk_access object x, invisibly, after printing its summary.

## print.walk_betweenness *Print Method for Walk Betweenness*

### Description

Print Method for Walk Betweenness

### Usage

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

### Arguments

| | |
|---|---|
| x | A walk_betweenness object. |
| ... | Additional arguments. |

### Value

The walk_betweenness object x, invisibly, after printing its summary.

## print.walk_comparison *Print Method for Walk Comparison*

### Description

Print Method for Walk Comparison

### Usage

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

### Arguments

| | |
|---|---|
| x | A walk_comparison object. |
| ... | Additional arguments. |

### Value

The walk_comparison object x, invisibly, after printing its summary.

## print.walk_diagnosis *Print Method for Street Network Diagnosis*

### Description

Print Method for Street Network Diagnosis

### Usage

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

### Arguments

| | |
|---|---|
| x | A walk_diagnosis object. |
| ... | Additional arguments. |

### Value

The walk_diagnosis object x, invisibly, after printing its summary.

## print.walk_isochrone *Print Method for Walk Isochrone*

### Description

Print Method for Walk Isochrone

### Usage

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

### Arguments

| | |
|---|---|
| x | A walk_isochrone object. |
| ... | Additional arguments. |

### Value

The walk_isochrone object x, invisibly, after printing its summary.

## print.walk_morphology *Print Method for Walk Morphology*

### Description

Print Method for Walk Morphology

### Usage

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

### Arguments

| | |
|---|---|
| x | A walk_morphology object. |
| ... | Additional arguments. |

### Value

The walk_morphology object x, invisibly, after printing its summary.

## print.walk_network *Print Method for Walk Network*

### Description

Print Method for Walk Network

### Usage

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

### Arguments

| | |
|---|---|
| x | A walk_network object. |
| ... | Additional arguments. |

### Value

The walk_network object x, invisibly, after printing its summary.

## print.walk_path *Print Method for Walk Path*

### Description

Print Method for Walk Path

### Usage

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

### Arguments

| | |
|---|---|
| x | A walk_path object. |
| ... | Additional arguments. |

### Value

The walk_path object x, invisibly, after printing its summary.

## print.walk_reach *Print Method for Walk Reach Metric*

### Description

Print Method for Walk Reach Metric

### Usage

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

### Arguments

| | |
|---|---|
| x | A walk_reach object. |
| ... | Additional arguments. |

### Value

The walk_reach object x, invisibly, after printing its summary.

## print.walk_scenario_betweenness *Print Method for Walk Scenario Betweenness*

### Description

Print Method for Walk Scenario Betweenness

### Usage

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

### Arguments

| | |
|---|---|
| x | A walk_scenario_betweenness object. |
| ... | Additional arguments. |

### Value

The walk_scenario_betweenness object x, invisibly, after printing its summary.

## rust_build_csr *Build Compressed Sparse Row (CSR) graph representation. Drops self-loops, collapses parallel edges to minimum travel time, and returns CSR structures along with connected component analysis.*

### Description

Build Compressed Sparse Row (CSR) graph representation. Drops self-loops, collapses parallel edges to minimum travel time, and returns CSR structures along with connected component analysis.

### Usage

```
rust_build_csr(n_nodes, u, v, wt, wl)
```

### Arguments

| | |
|---|---|
| n_nodes | Number of nodes in the graph (0-indexed: 0..n-1). |
| u | From-node vector (0-indexed). |
| v | To-node vector (0-indexed). |
| wt | Travel time in seconds for each edge. |
| wl | Physical length in meters for each edge. |

## Value

A named `list` containing CSR arrays `head`, `to`, `wt`, and `wl`; node component IDs `comp`; and graph summaries `n_edges_collapsed`, `giant_comp_id`, `giant_comp_size`, and `total_components`. Node and component IDs are zero based.

---

`rust_compute_access_multi` *Compute cumulative POI accessibility for target nodes in parallel.*

---

## Description

Compute cumulative POI accessibility for target nodes in parallel.

## Usage

```
rust_compute_access_multi(
  n_nodes,
  head_num,
  to,
  wt,
  target_nodes,
  poi_nodes,
  poi_weights,
  cutoff,
  num_threads
)
```

## Arguments

| | |
|---|---|
| `n_nodes` | Total number of nodes. |
| `head_num` | CSR head array. |
| `to` | CSR destination array. |
| `wt` | CSR travel time array (seconds). |
| `target_nodes` | Indices of nodes to compute accessibility for (0-indexed). |
| `poi_nodes` | Vector of node indices where POIs are situated. |
| `poi_weights` | Vector of weights/counts for each POI node. |
| `cutoff` | Travel time cutoff in seconds. |
| `num_threads` | Number of worker threads (1 or 2; values outside this range are clamped to the supported range by the native implementation). |

## Value

A named `list` with numeric `accessible_count` and `min_time_s` vectors, one element per target node. These give weighted reachable POI counts and travel time in seconds to the nearest reachable POI, respectively.

`rust_compute_circuity_sample` *Compute detour ratio (circuity) for sampled origin-destination pairs.*

## Description

Circuity is defined as actual network shortest path distance divided by Euclidean straight-line distance. Values close to 1.0 indicate highly direct routes; higher values indicate circuitous paths and poor permeability.

## Usage

```
rust_compute_circuity_sample(
  n_nodes,
  head_num,
  to,
  wt,
  wl,
  coords_x,
  coords_y,
  sample_nodes,
  cutoff
)
```

## Arguments

| | |
|---|---|
| `n_nodes` | Total number of nodes. |
| `head_num` | CSR head array. |
| `to` | CSR destination array. |
| `wt` | CSR travel time array. |
| `wl` | CSR length array. |
| `coords_x` | Vector of node X coordinates (meters). |
| `coords_y` | Vector of node Y coordinates (meters). |
| `sample_nodes` | Nodes from which to measure circuity. |
| `cutoff` | Travel time cutoff in seconds. |

## Value

A named `list` with `node_id`, `mean_circuity`, `max_circuity`, and `destinations_reached` vectors. Each position summarizes the network to straight-line distance ratios for one sampled origin.

`rust_compute_reachable_length` *Compute reachable walkable network length in parallel for a set of target nodes.*

## Description

Compute reachable walkable network length in parallel for a set of target nodes.

## Usage

```
rust_compute_reachable_length(
  n_nodes,
  head_num,
  to,
  wt,
  wl,
  target_nodes,
  cutoff,
  num_threads
)
```

## Arguments

| | |
|---|---|
| `n_nodes` | Total number of nodes in graph. |
| `head_num` | CSR head array. |
| `to` | CSR destination nodes. |
| `wt` | CSR travel times (seconds). |
| `wl` | CSR lengths (meters). |
| `target_nodes` | Indices of nodes to compute metric for (0-indexed). |
| `cutoff` | Walk travel-time budget (seconds, default 600.0 = 10 min). |
| `num_threads` | Number of worker threads (1 or 2; values outside this range are clamped to the supported range by the native implementation). |

## Value

A numeric vector, one value per `target_nodes` entry, giving reachable street length in meters within `cutoff`. Invalid target nodes yield `NaN`.

`rust_isochrone_subgraph` *Extract reachable edges and nodes within cutoff time from an origin node.*

## Description

Extract reachable edges and nodes within cutoff time from an origin node.

## Usage

```
rust_isochrone_subgraph(n_nodes, head_num, to, wt, wl, src_node, cutoff)
```

## Arguments

| | |
|---|---|
| `n_nodes` | Total number of nodes. |
| `head_num` | CSR head array. |
| `to` | CSR destination array. |
| `wt` | CSR travel time array (seconds). |
| `wl` | CSR length array (meters). |
| `src_node` | Origin node (0-indexed). |
| `cutoff` | Travel time cutoff in seconds. |

## Value

A named `list` with `reached_nodes` and their `node_times`, plus `edge_from`, `edge_to`, `edge_fraction`, `edge_length`, and `edge_reached_length` describing reachable portions of street edges. Node IDs are zero based, times are seconds, and lengths are meters.

`rust_shortest_path` *Compute shortest path between source and target nodes using Dijkstra.*

## Description

Compute shortest path between source and target nodes using Dijkstra.

## Usage

```
rust_shortest_path(n_nodes, head_num, to, wt, wl, src_node, dst_node)
```

### Arguments

| | |
|---|---|
| n_nodes | Total number of nodes. |
| head_num | CSR head array. |
| to | CSR destination array. |
| wt | CSR travel time array (seconds). |
| wl | CSR length array (meters). |
| src_node | Origin node (0-indexed). |
| dst_node | Destination node (0-indexed). |

### Value

A named list with logical found, zero-based node_path and edge_indices vectors, and numeric total_time_s and total_length_m. When no path is found, paths are empty and totals are NaN.

---

## streets_arequipa *Walkable Street Network of Arequipa (Historic Center / Yanahuara)*

---

### Description

A spatial sf line dataset representing the walkable street network of the Historic Center, Plaza de Armas, and Yanahuara in Arequipa, Peru. Extracted from OpenStreetMap and projected into UTM Zone 19S (EPSG:32719).

### Usage

```
streets_arequipa
```

### Format

An sf data frame with LINESTRING geometries and columns:

**street_id** Unique integer identifier for each street segment.

**osm_id** Original OpenStreetMap way identifier.

**name** Street or passage name (if available).

**highway** OpenStreetMap road classification (e.g. pedestrian, footway, residential, tertiary).

**oneway** Directionality tag from OSM.

**surface** Pavement surface type.

**geom** LineString geometry in EPSG:32719.

### Source

OpenStreetMap contributors (https://www.openstreetmap.org/copyright), ODbL 1.0.

### Examples

```
data(streets_arequipa)
nrow(streets_arequipa)
```

## streets_cusco *Walkable Street Network of Cusco (Historic Center)*

### Description

A spatial `sf` line dataset representing the walkable street network of the Historic Center and San Blas district in Cusco, Peru. Extracted from OpenStreetMap and projected into UTM Zone 19S (EPSG:32719).

### Usage

```
streets_cusco
```

### Format

An `sf` data frame with `LINESTRING` geometries and columns:

**street_id** Unique integer identifier for each street segment.

**osm_id** Original OpenStreetMap way identifier.

**name** Street or passage name (if available).

**highway** OpenStreetMap road classification (e.g. `pedestrian`, `footway`, `residential`, `steps`).

**oneway** Directionality tag from OSM.

**surface** Pavement surface type (e.g. `paved`, `cobblestone`).

**geom** LineString geometry in EPSG:32719.

### Source

OpenStreetMap contributors (https://www.openstreetmap.org/copyright), ODbL 1.0.

### Examples

```
data(streets_cusco)
nrow(streets_cusco)
```

## streets_lima *Walkable Street Network of Lima (Historic Center / Damero de Pizarro)*

### Description

A spatial `sf` line dataset representing the walkable street network of the Historic Center and Damero de Pizarro in Lima, Peru. Extracted from OpenStreetMap and projected into UTM Zone 18S (EPSG:32718).

### Usage

```
streets_lima
```

### Format

An sf data frame with LINESTRING geometries and columns:

**street_id** Unique integer identifier for each street segment.

**osm_id** Original OpenStreetMap way identifier.

**name** Street or passage name (if available).

**highway** OpenStreetMap road classification (e.g. pedestrian, footway, residential, secondary).

**oneway** Directionality tag from OSM.

**surface** Pavement surface type.

**geom** LineString geometry in EPSG:32718.

### Source

OpenStreetMap contributors (https://www.openstreetmap.org/copyright), ODbL 1.0.

### Examples

```
data(streets_lima)
nrow(streets_lima)
```

---

## theme_walkability *Modern Minimalist Cartographic Theme for Walkability*

---

### Description

Clean theme optimized for geospatial network visualizations with ggplot2.

### Usage

```
theme_walkability(base_size = 11, base_family = "", dark = TRUE)
```

### Arguments

| | |
|---|---|
| base_size | Base font size. |
| base_family | Base font family. |
| dark | Logical. If TRUE (default), uses a sleek dark theme. If FALSE, uses a clean white theme. |

### Value

A ggplot2 theme object that styles plot backgrounds, text, and legends for walkability maps.

walk_app *Launch the Interactive Walkability Application*

## Description

Launches an interactive web application built with Shiny, `bslib`, and `mapgl` (MapLibre GL JS) for visual exploration and quantitative analysis of urban street networks.

## Usage

```
walk_app(
  network = NULL,
  host = "127.0.0.1",
  port = NULL,
  launch.browser = TRUE,
  lang = c("es", "en"),
  ...
)
```

## Arguments

| | |
|---|---|
| `network` | Optional initial `walk_network` object. If `NULL` (default), the app allows selecting among the built-in model cities (Cusco, Lima, Arequipa) or uploading a custom street dataset. |
| `host` | Character. The IPv4 address that the application should listen on (default `"127.0.0.1"`). |
| `port` | Integer. The TCP port that the application should listen on. Default is randomly selected. |
| `launch.browser` | Logical. Whether to open the application in the system's default browser (default `TRUE`). |
| `lang` | Character string specifying the initial interface language: `"es"` for Spanish (default) or `"en"` for English. The user can also toggle the language interactively at runtime via the navbar selector. |
| `...` | Additional arguments passed to `shiny::runApp()`. |

## Details

The application provides dedicated, fully documented workspaces for:

- **Pedestrian Reachability (10-min Reach):** Spatial gradient of walkable network length with partial-edge credits in Rust.
- **Multi-Threshold Isochrones:** Concentric, non-overlapping topological rings (e.g. 5, 10, 15 min) or cumulative hulls.
- **Point-to-Point Routing & Circuity:** Optimal shortest path, travel time, and detour ratio.
- **Network Morphology & Intersection Density:** Topological node valence classification (cul-de-sacs, 3-way, 4-way+).

- **Scenario Simulation:** Testing street closures and pedestrianization with before/after comparison.
- **Documentation & Code Generator:** Detailed methodology and reproducible R scripts.

## Value

Runs the Shiny application (returns invisible NULL).

## Examples

```
if (interactive()) {
# Launch with default language (Spanish):
walk_app()

# Launch in English:
walk_app(lang = "en")

# Launch with a pre-built network:
utils::data(streets_cusco)
net <- walk_build_network(streets_cusco, verbose = FALSE)
walk_app(net, lang = "es")
}
```

---

# walk_betweenness *Street Network Edge Betweenness Centrality*

---

## Description

Computes the edge betweenness centrality for all street segments in a walkable network. Betweenness measures the proportion of potential shortest paths that traverse each street segment. A high value denotes a critical bridge, connector, or structural bottleneck: if interrupted, many potential routes are forced to divert.

## Usage

```
walk_betweenness(
  network,
  snap_tolerance_m = 0,
  radius_m = NULL,
  weights = "length_m",
  directed = FALSE,
  normalized = TRUE
)
```

## Arguments

| | |
|---|---|
| network | A walk_network object produced by walk_build_network. |
| snap_tolerance_m | Numeric. Radius in meters for network gap correction (snapping tolerance). If > 0, repairs unconnected endpoints before graph construction. Default is 0.0. |
| radius_m | Optional numeric. Maximum metric search radius (cutoff) for shortest paths. If NULL or <= 0, calculates global betweenness across the entire connected network. |
| weights | Character string naming the edge attribute used for path lengths. Default is "length_m". |
| directed | Logical. If TRUE, considers directed edges; default is FALSE (bidirectional pedestrian walkability). |
| normalized | Logical. If TRUE, adds a normalized score (0 to 100) and percentile rank. Default is TRUE. |

## Details

**What Edge Betweenness Measures:** The betweenness centrality of an edge $e$ is defined as:

$$C_B(e) = \sum_{s \neq t} \frac{\sigma_{st}(e)}{\sigma_{st}}$$

where $\sigma_{st}$ is the total number of shortest paths from node $s$ to node $t$, and $\sigma_{st}(e)$ is the number of those paths that pass through edge $e$.

**Important Methodological Note:** Betweenness is a purely structural topological metric. It is **not** a traffic volume count (aforo) nor a congestion forecast. The tool does not assume vehicle volumes, origin-destination matrices, travel schedules, one-way regulations, or actual speeds. Consequently, high-betweenness (highlighted in red) street segments represent structural network priorities, but should not be conflated with "streets carrying the most traffic".

**Network Correction Radius:** If snap_tolerance_m > 0, street segments with small digitization gaps or undershoots within that metric tolerance are snapped and reconciled before calculating shortest-path flow, avoiding artificial topological breaks.

## Value

A walk_betweenness S3 object containing:

| | |
|---|---|
| edges | sf LINESTRING data frame with columns betweenness, betweenness_norm, betweenness_pctl, and bottleneck_class. |
| summary | Summary statistics of betweenness across the network. |
| top_bottlenecks | Top street segments with highest betweenness. |
| network | The evaluated walk_network. |
| snap_tolerance_m | The snapping tolerance used. |
| radius_m | The cutoff radius used. |

## Examples

```
if (requireNamespace("igraph", quietly = TRUE)) {
  data(streets_cusco)
  net <- walk_build_network(streets_cusco, verbose = FALSE)
  bw <- walk_betweenness(net)
  print(bw)
}
```

---

`walk_build_network` *Build Walkable Graph Network from Street Data*

---

## Description

Converts user-supplied or downloaded spatial street data into a clean, pedestrian-routing graph structure. Computes edge travel times at a constant walking speed, optionally performs geometric repairs (planarization, endpoint snapping) at the user's discretion, and prepares Compressed Sparse Row (CSR) topology via compiled Rust.

## Usage

```
walk_build_network(
  data,
  crs = NULL,
  walk_speed_kmh = 4.5,
  repair = FALSE,
  planarize = FALSE,
  snap_tolerance_m = 0,
  keep_components = c("largest", "all"),
  drop_classes = c("motorway", "trunk"),
  drop_subclasses = TRUE,
  verbose = TRUE
)
```

## Arguments

| | |
|---|---|
| `data` | An `sf` object with `LINESTRING` or `MULTILINESTRING` geometries. |
| `crs` | Optional target metric CRS (e.g. `EPSG:32722` or `"auto"` to automatically infer the appropriate local UTM zone). If provided and `data` is in a different CRS, it will be projected. |
| `walk_speed_kmh` | Walking design speed in km/h (default is 4.5 km/h, the international pedestrian planning standard, approx. 1.25 m/s). |
| `repair` | Logical. If `TRUE`, applies the requested topological cleaning (`planarize`, `snap_tolerance_m`). If `FALSE`, preserves geometry exactly as provided. |
| `planarize` | Logical. If `TRUE` and `repair = TRUE`, splits crossing lines at intersections to create valid routing junctions. |

snap_tolerance_m
: Numeric. Threshold in meters to snap endpoint gaps. Default is 0.0 (no snapping). A value between 2.0 and 5.0 is recommended for imperfect street databases.

keep_components
: Character. Either "largest" (default) to extract the main connected pedestrian network, or "all" to retain disconnected fragments.

drop_classes
: Character vector of road classes to exclude (e.g. c("motorway", "trunk")). Only applied if a class or highway column is present in data.

drop_subclasses
: Logical. If TRUE, drops non-network subclasses like sidewalk, crosswalk, cycle_crossing, driveway, and parking_aisle if metadata exists.

verbose
: Logical. If TRUE, prints a summary of network construction.

## Value

A walk_network S3 object containing:

nodes
: sf point collection with node coordinates and IDs.

edges
: sf line collection with length_m, travel_time_s, from-node, to-node.

csr
: List with Compressed Sparse Row topology arrays (head, to, wt, wl, comp).

walk_speed_kmh
: The walking speed used.

giant_comp_id
: ID of the giant connected component.

crs
: The coordinate reference system of the network.

## Examples

```
data(streets_cusco)
net <- walk_build_network(streets_cusco, verbose = FALSE)
print(net)
```

---

# walk_compare_scenarios

*Compare Baseline and Intervention Scenarios*

---

## Description

Compares structural topology and reachable walkable network length between a baseline street network and an intervention scenario (e.g. street closures).

## Usage

```
walk_compare_scenarios(base, scenario, time_min = 10, verbose = TRUE)
```

### Arguments

| | |
|---|---|
| base | Baseline walk_network object. |
| scenario | Modified walk_network (or walk_scenario) object. |
| time_min | Walking budget in minutes for reach comparison (default 10.0 min). |
| verbose | Logical. If TRUE, prints comparison summary. |

### Value

A walk_comparison S3 object.

### Examples

```
data(streets_cusco)
net <- walk_build_network(streets_cusco, verbose = FALSE)
cierre <- walk_scenario_close(net, edge_indices = c(5, 12), name = "Peatonalizacion Central")
comp <- walk_compare_scenarios(base = net, scenario = cierre, time_min = 5, verbose = FALSE)
print(comp)
```

---

## walk_compute_access *Compute Cumulative Walkable Accessibility to Points of Interest (POIs)*

---

### Description

Computes cumulative pedestrian accessibility from each network junction to urban facilities (e.g. schools, health centers, pharmacies, public transit stops) within a walking time budget (default 10 minutes at 4.5 km/h). Uses parallel compiled Rust to evaluate hundreds of thousands of shortest paths across CPU cores.

### Usage

```
walk_compute_access(
  network,
  pois,
  time_min = 10,
  category_col = NULL,
  weight_col = NULL,
  target = c("largest_component", "all"),
  n_threads = 2L,
  verbose = TRUE
)
```

### Arguments

| | |
|---|---|
| network | A walk_network object produced by walk_build_network(). |
| pois | An sf points collection containing destination facilities/POIs. |
| time_min | Walking time threshold in minutes (default 10.0 minutes). |
| category_col | Optional column name in pois indicating facility category (e.g. "amenity", "category", or "type"). If provided, calculates separate accessibility counts for each unique category. |
| weight_col | Optional column name in pois indicating facility weight/capacity. Default is NULL (each POI counts as 1.0). |
| target | Character string: "largest_component" (default) to evaluate nodes in the primary connected component, or "all". |
| n_threads | Number of CPU threads to use in parallel, from 1 to 2. The default is 2, the maximum permitted during CRAN checks. |
| verbose | Logical. If TRUE, prints execution progress and summary. |

### Value

A walk_access S3 object containing:

| | |
|---|---|
| nodes | sf points table augmented with columns for cumulative counts and minimum walking times. |
| summary | Summary statistics table per category. |
| time_min | The evaluated walking budget in minutes. |
| pois | The original POI points mapped to network nodes. |
| network | The input walk_network. |

### Examples

```
data(streets_cusco)
net <- walk_build_network(streets_cusco, verbose = FALSE)
sample_pois <- net$nodes[net$nodes$is_giant, ][seq(1, 100, by = 20), ]
acc <- walk_compute_access(net, pois = sample_pois, time_min = 5, verbose = FALSE)
print(acc)
```

---

## walk_compute_circuity *Compute Network Circuity (Detour Ratio) for Sampled Junctions*

---

### Description

Evaluates path directness by comparing network shortest-path distances against Euclidean straight-line distances. High circuity signifies indirect, circuitous street patterns and cul-de-sacs.

### Usage

```
walk_compute_circuity(
  network,
  sample_size = 500,
  time_min = 10,
  cutoff_s = NULL
)
```

### Arguments

| | |
|---|---|
| network | A walk_network object. |
| sample_size | Number of random junctions to sample. Default is 500. |
| time_min | Walking time budget in minutes (default is 10.0 minutes). |
| cutoff_s | Optional walking time budget in seconds (default is NULL). If provided, it overrides time_min. |

### Value

A tibble with node IDs and calculated detour ratios.

---

## walk_compute_reach *Compute Reachable Walkable Network Length Metric*

---

### Description

Computes the total walkable network length lying within each node's isochrone (default 10-minute walk at 4.5 km/h, ~750 m budget). The calculation runs in parallel using compiled Rust and applies the exact partial-edge credit rule, crediting crossing frontier edges for only their reachable fraction.

### Usage

```
walk_compute_reach(
  network,
  time_min = 10,
  cutoff_s = NULL,
  target = c("largest_component", "all"),
  n_threads = 2L,
  verbose = TRUE
)
```

### Arguments

| | |
|---|---|
| network | A walk_network object produced by walk_build_network(). |
| time_min | Walking time budget in minutes (default is 10.0 minutes). Users can freely customize this parameter; for smaller neighborhoods, compact districts, or localized analyses, values like 5.0 or 7.5 minutes can be used. |

| | |
|---|---|
| cutoff_s | Optional walking time budget in seconds (e.g. 600.0). If specified, it overrides time_min. |
| target | Character string. Either "largest_component" (default) to compute only for nodes in the primary connected component, or "all". |
| n_threads | Number of CPU threads to use in parallel, from 1 to 2. The default is 2, the maximum permitted during CRAN checks. |
| verbose | Logical. If TRUE, prints summary percentiles and runtime. |

## Value

A walk_reach S3 object containing:

| | |
|---|---|
| nodes | sf points table augmented with reachable_length_m and reachable_length_km. |
| edges | sf lines table augmented with reachable_length_km (mean of endpoints). |
| summary | Summary statistics (min, mean, max, 5th, 95th percentiles). |
| time_min | The time cutoff in minutes. |
| cutoff_s | The time cutoff in seconds. |
| network | The input walk_network object. |

## Examples

```
data(streets_cusco)
net <- walk_build_network(streets_cusco, verbose = FALSE)
reach <- walk_compute_reach(net, time_min = 5, verbose = FALSE)
print(reach)
```

---

# walk_diagnose_network *Topological and Health Diagnosis for Street Networks*

---

## Description

Evaluates the topological connectivity, dangling nodes, potential undershoots, and component fragmentation of a user-supplied street network before building the routing graph. This function does not alter geometries, leaving repair decisions to the user.

## Usage

```
walk_diagnose_network(data, snap_search_radius_m = 10, crs = NULL)
```

## Arguments

| | |
|---|---|
| data | An sf object containing LINESTRING or MULTILINESTRING street segments. |
| snap_search_radius_m | Maximum distance in meters to search for potential undershoots (unconnected ends located near another segment). Default is 10.0 meters. |
| crs | Optional target metric CRS (e.g. EPSG code or proj4string). If provided, data is transformed to this CRS before distance calculations. |

## Details

**Concept: Undershoots:** An **undershoot** occurs when a digitized street segment terminates just short of intersecting or connecting with an adjacent road, leaving a small metric gap. While legitimate dead ends (cul-de-sacs) naturally exist in pedestrian networks, a dangling endpoint that lies suspiciously close (within `snap_search_radius_m`) to another street is typically a digitization or vectorization defect.

In `walk_diagnose_network()`, undershoots are identified in two stages:

1. All dangling endpoints (nodes with topological degree = 1) are isolated across the network.
2. The minimum Euclidean distance from each dangling endpoint to all other line geometries (excluding the incident segment itself) is computed. If this distance is less than or equal to `snap_search_radius_m`, the endpoint is flagged as a potential undershoot.

In pedestrian routing, even a gap of a few centimeters forms an impassable barrier, artificially forcing detours or disconnecting origins from destinations. Potential undershoots can be resolved during network construction via `walk_build_network(..., repair = TRUE, snap_tolerance_m = ...)`.

**Concept: Isolated Edges:** **Isolated edges** are street segments that belong to disconnected components (subgraphs or "islands") outside of the primary **giant component** (the largest contiguous connected network).

In `walk_diagnose_network()`, connectivity components are detected using a fast disjoint-set (Union-Find) algorithm implemented in Rust:

1. Endpoints are matched across lines using coordinate coincidence with 1 mm precision.
2. The giant component is determined as the component with the largest total network length.
3. Any edge where neither endpoint connects to the giant component is labeled as an isolated edge.

Isolated fragments commonly arise from bounding box or polygon clipping (cutting streets at administrative borders), gated communities, or disjointed municipal data layers. When routing or computing isochrones, any origin or destination snapped to an isolated edge will be mathematically unreachable from the rest of the city (resulting in infinite travel times).

**Visual Inspection:** Diagnostic results can be inspected visually using `plot(diagnosis)` or `walk_plot_diagnosis()`, which renders the giant component in neutral tones, highlights isolated edges in red, and marks potential undershoots with amber points.

## Value

A `walk_diagnosis` S3 object containing:

| | |
|---|---|
| `summary` | Tibble with diagnostic statistics (segments, lengths, components, dangles, undershoots). |
| `components` | Table of detected connected components and their total lengths. |
| `undershoots` | `sf` point collection of suspicious dangling endpoints within `snap_search_radius_m` of another street, with column `dist_to_nearest_m`. |
| `isolated_edges` | `sf` line collection of edges not belonging to the giant component. |
| `data` | Original spatial line data. |
| `search_radius_m` | The search radius parameter used. |

## See Also

walk_plot_diagnosis(), walk_build_network()

## Examples

```
data(streets_cusco)
diag <- walk_diagnose_network(streets_cusco, snap_search_radius_m = 5)
print(diag)
```

---

# walk_download_osm *Download OpenStreetMap Street Network Data*

---

## Description

Downloads walkable street networks using OpenStreetMap via the osmdata package.

## Usage

```
walk_download_osm(place_name, buffer_m = 2000)
```

## Arguments

| | |
|---|---|
| place_name | Character. Geographic name to query (e.g. "Curitiba, Brazil"). |
| buffer_m | Buffer distance in meters around the bounding box. Default is 2000 meters. |

## Value

An sf data frame with line geometries.

---

# walk_intersection_density *Compute Intersection Density and Street Network Morphology*

---

## Description

Evaluates the structural morphology of a walkable street network by analyzing node valence (degrees), isolating cul-de-sacs (dead ends), counting 3-way (T-junctions) and 4-way+ (crossroads) intersections, and calculating density metrics.

## Usage

```
walk_intersection_density(network, study_area = NULL)
```

### Arguments

| | |
|---|---|
| network | A walk_network object produced by walk_build_network(). |
| study_area | Optional sf polygon defining the bounding study area. If NULL, the convex hull of the street network is used to compute area in km^2. |

### Value

A walk_morphology S3 object containing:

| | |
|---|---|
| summary | A named list of summary morphological metrics. |
| nodes | sf points table augmented with node degree and junction_type. |
| network | The input walk_network. |

### Examples

```
data(streets_cusco)
net <- walk_build_network(streets_cusco, verbose = FALSE)
morph <- walk_intersection_density(net)
print(morph)
```

---

## walk_isochrone *Compute Pedestrian Isochrone Polygons and Reachable Subgraphs*

---

### Description

Generates spatial polygon isochrones (concave hulls or buffered envelopes) and extracts the exact reachable street network from an origin location for specified walking time budgets (e.g. 5, 10, 15 minutes).

### Usage

```
walk_isochrone(
  network,
  origin,
  time_min = c(5, 10, 15),
  method = c("concave", "buffer"),
  buffer_m = 25,
  concave_ratio = 0.3,
  rings = TRUE
)
```

### Arguments

| | |
|---|---|
| network | A walk_network object produced by walk_build_network(). |
| origin | Origin location: an integer node ID, coordinate vector c(x, y), or an sf POINT. |
| time_min | Numeric vector of walking time thresholds in minutes (e.g. c(5, 10, 15) or 10.0). |
| method | Polygon generation technique: "concave" for concave hull envelopes or "buffer" for street network buffering. |
| buffer_m | Buffer width in meters applied along reachable edges when synthesizing pedestrian walkable corridors (default 25.0 m). |
| concave_ratio | Ratio parameter passed to sf::st_concave_hull() (0 to 1, default 0.3). |
| rings | Logical. If TRUE (default), multi-threshold polygons are transformed into mutually disjoint, non-overlapping concentric rings (e.g. 0-5 min, 5-10 min, 10-15 min) via topological differences, avoiding polygon occlusion. Cumulative envelopes are still preserved in the cumulative_polygons element. |

### Value

A walk_isochrone S3 object containing:

| | |
|---|---|
| polygons | sf polygons representing the isochrones (rings or cumulative). |
| cumulative_polygons | sf polygons with full cumulative envelopes. |
| edges | sf lines of all reached street segments with travel time. |
| origin | sf point of the resolved origin node. |
| time_min | The evaluated time thresholds. |
| rings | Logical indicating whether disjoint rings were generated. |
| network | The input walk_network. |

### Examples

```
data(streets_cusco)
net <- walk_build_network(streets_cusco, verbose = FALSE)
orig <- sf::st_coordinates(net$nodes[net$nodes$is_giant, ])[1, 1:2]
iso <- walk_isochrone(net, origin = orig, time_min = c(5, 10), rings = TRUE)
print(iso)
```

---

## walk_plot_diagnosis *Plot Street Network Topological Diagnosis*

---

### Description

Maps the health of a street network, displaying the main connected component in neutral tones and highlighting isolated fragments and potential undershoots.

### Usage

```
walk_plot_diagnosis(
  diagnosis,
  title = "Street Network Topological Health",
  dark = TRUE
)
```

### Arguments

| | |
|---|---|
| diagnosis | A walk_diagnosis object produced by walk_diagnose_network(). |
| title | Optional custom title. |
| dark | Logical. If TRUE (default), uses dark theme. |

### Value

A ggplot object.

---

## walk_plot_reach *Plot Reachable Walkable Network Length Map*

---

### Description

Generates a publication-grade cartographic visualization of reachable walkable network length using ggplot2. Segments are rendered in ascending order so dense, high-connectivity areas (yellow) plot on top. The color scale is clipped to the 2nd-98th percentiles with saturation arrows for optimal visual contrast.

### Usage

```
walk_plot_reach(
  reach,
  boundary = NULL,
  palette = "viridis",
  p_clip = c(0.02, 0.98),
  title = NULL,
  subtitle = NULL,
  line_width = 0.4,
  dark = TRUE
)
```

### Arguments

| | |
|---|---|
| reach | A walk_reach object produced by walk_compute_reach(). |
| boundary | Optional sf polygon representing the municipal or study boundary. |
| palette | Palette name for ggplot2::scale_color_viridis_c (default "viridis"). |

| | |
|---|---|
| p_clip | Percentile range for color clipping (default c(0.02, 0.98)). |
| title | Optional custom title. Default highlights the isochrone budget. |
| subtitle | Optional subtitle. |
| line_width | Width of street lines. Default is 0.4. |
| dark | Logical. If TRUE (default), renders on a dark aesthetic canvas. |

## Value

A ggplot object.

---

walk_scenario_betweenness *Simulate Scenario Closures and Evaluate Betweenness Re-routing*

---

## Description

Simulates the pedestrianization or physical closure of selected street segments and calculates the redistribution of shortest-path betweenness centrality across the remaining network.

## Usage

```
walk_scenario_betweenness(
  network,
  closed_edges = NULL,
  snap_tolerance_m = 0,
  radius_m = NULL,
  verbose = TRUE
)
```

## Arguments

| | |
|---|---|
| network | A walk_network object. |
| closed_edges | Integer vector of edge row indices to remove. |
| snap_tolerance_m | Snapping tolerance in meters for network repair. Default is 0.0. |
| radius_m | Optional metric search cutoff in meters. |
| verbose | Logical. |

## Value

A walk_scenario_betweenness S3 object containing comparison metrics, augmented edges with delta betweenness, and top diverted streets.

## Examples

```
if (requireNamespace("igraph", quietly = TRUE)) {
  data(streets_cusco)
  net <- walk_build_network(streets_cusco, verbose = FALSE)
  sim <- walk_scenario_betweenness(net, closed_edges = c(5, 10))
  print(sim)
}
```

---

`walk_scenario_close` *Simulate Street Closure or Pedestrianization Intervention Scenario*

---

## Description

Creates an independent copy of a walkable street network with specified street segments (edges) or junctions (nodes) removed (simulating street cuts, construction, or pedestrianization). Reconstructs Compressed Sparse Row topology and component connectivity using compiled Rust.

## Usage

```
walk_scenario_close(
  network,
  edge_indices = NULL,
  node_indices = NULL,
  name = "scenario_closure"
)
```

## Arguments

| | |
|---|---|
| `network` | A `walk_network` object produced by `walk_build_network()`. |
| `edge_indices` | Integer vector of edge row indices to remove (1-indexed). |
| `node_indices` | Optional integer vector of node indices whose attached edges will be removed. |
| `name` | Descriptive character label for this intervention scenario. |

## Value

A new `walk_network` S3 object representing the modified scenario, with metadata storing the closed edges.

## Examples

```
data(streets_cusco)
net <- walk_build_network(streets_cusco, verbose = FALSE)
cierre <- walk_scenario_close(net, edge_indices = c(5, 12), name = "Peatonalizacion Central")
print(cierre)
```

walk_shortest_path *Compute Shortest Pedestrian Path Between Two Locations*

## Description

Finds the optimal walking route between an origin and destination across the street graph using Dijkstra's algorithm compiled in Rust. Supports node IDs, numeric coordinate vectors `c(x, y)`, or `sf` point geometries.

## Usage

```
walk_shortest_path(network, from, to)
```

## Arguments

| | |
|---|---|
| `network` | A `walk_network` object created by `walk_build_network()`. |
| `from` | Origin location: either an integer node index, a numeric vector `c(x, y)` in the network's metric CRS, or an `sf` POINT. |
| `to` | Destination location: either an integer node index, a numeric vector `c(x, y)` in the network's metric CRS, or an `sf` POINT. |

## Value

A `walk_path` S3 object containing:

| | |
|---|---|
| `path_sf` | `sf` object with the route geometry (`LINESTRING`) and attributes. |
| `nodes_sf` | `sf` points along the path in traversed order. |
| `node_ids` | Sequence of node IDs visited. |
| `distance_m` | Total walking route distance in meters. |
| `time_s` | Total travel time in seconds at the network walking speed. |
| `time_min` | Total travel time in minutes. |
| `circuity` | Detour ratio (network path distance / straight-line Euclidean distance). |
| `from_node` | Resolved origin node index. |
| `to_node` | Resolved destination node index. |
| `network` | The input `walk_network`. |

## Examples

```
data(streets_cusco)
net <- walk_build_network(streets_cusco, verbose = FALSE)
orig <- sf::st_coordinates(net$nodes[net$nodes$is_giant, ])[1, 1:2]
dest <- sf::st_coordinates(net$nodes[net$nodes$is_giant, ])[50, 1:2]
p <- walk_shortest_path(net, from = orig, to = dest)
print(p)
```

# Index


∗ **datasets**
    streets_arequipa, 18
    streets_cusco, 19
    streets_lima, 19

as_sfnetwork, 3
as_tidygraph, 4

plot.walk_access, 4
plot.walk_betweenness, 5
plot.walk_comparison, 5
plot.walk_diagnosis, 6
plot.walk_isochrone, 6
plot.walk_morphology, 7
plot.walk_path, 7
plot.walk_reach, 8
print.walk_access, 8
print.walk_betweenness, 9
print.walk_comparison, 9
print.walk_diagnosis, 10
print.walk_isochrone, 10
print.walk_morphology, 11
print.walk_network, 11
print.walk_path, 12
print.walk_reach, 12
print.walk_scenario_betweenness, 13

rust_build_csr, 13
rust_compute_access_multi, 14
rust_compute_circuity_sample, 15
rust_compute_reachable_length, 16
rust_isochrone_subgraph, 17
rust_shortest_path, 17

sf::st_concave_hull(), *33*
shiny::runApp(), *21*
streets_arequipa, 18
streets_cusco, 19
streets_lima, 19

theme_walkability, 20

walk_app, 21
walk_betweenness, 22
walk_build_network, *23*, 24
walk_build_network(), *27, 28*, *31–33*, *36, 37*
walk_compare_scenarios, 25
walk_compute_access, 26
walk_compute_circuity, 27
walk_compute_reach, 28
walk_compute_reach(), *34*
walk_diagnose_network, 29
walk_diagnose_network(), *34*
walk_download_osm, 31
walk_intersection_density, 31
walk_isochrone, 32
walk_plot_diagnosis, 33
walk_plot_diagnosis(), *30, 31*
walk_plot_reach, 34
walk_plot_reach(), *8*
walk_scenario_betweenness, 35
walk_scenario_close, 36
walk_shortest_path, 37