

Package: sfnetworks (via r-universe)

October 15, 2026

Title Tidy Geospatial Networks

Version 0.9.9.9000

Maintainer Lucas van der Meer <luukvandermeer@live.nl>

Description Provides a tidy approach to spatial network analysis, in the form of classes and functions that enable a seamless interaction between the network analysis package 'tidygraph' and the spatial analysis package 'sf'.

License Apache License (>= 2)

URL <https://luukvdmeer.github.io/sfnetworks/>,
<https://github.com/luukvdmeer/sfnetworks>

BugReports <https://github.com/luukvdmeer/sfnetworks/issues/>

Depends R (>= 3.6)

Imports cli (>= 3.0.0), dplyr (>= 1.1.0), graphics, igraph (>= 2.1.0), lifecycle (>= 1.0.0), lwgeom, methods, pillar, rlang (>= 1.0.0), sf (>= 1.0-11), sfheaders (>= 0.2.2), stats, tibble, tidygraph (>= 1.3.0), tidyselect (>= 1.0.0), units (>= 0.4.6), utils

Suggests dbscan, dodgr, fansi, geodist, ggplot2 (>= 3.0.0), ggraph (>= 2.2.0), knitr, osmdata, purrr, quarto, rmarkdown, s2 (>= 1.0.1), spatstat.geom, spatstat.linnet, spdep, testthat, TSP

VignetteBuilder knitr, quarto

ByteCompile true

Encoding UTF-8

LazyData true

RoxygenNote 7.3.2

Config/pak/sysreqs libabsl-dev cmake libgdal-dev gdal-bin libgeos-dev libglpk-dev libicu-dev libxml2-dev libssl-dev libproj-dev libsqlite3-dev libudunits2-dev

Repository <https://luukvdmeer.r-universe.dev>

Date/Publication 2026-05-14 15:15:18 UTC

RemoteUrl <https://github.com/luukvdmeer/sfnetworks>

RemoteRef HEAD

RemoteSha fe6edf8c9a73160110cbf25b673853b7aa23f927

Contents

as.linnet	3
as_s2_geography	4
as_sfnetwork	4
as_tibble	7
autoplot	8
bind_spatial	9
contract_nodes	10
create_from_spatial_lines	11
create_from_spatial_points	12
data	15
dual_weights	16
evaluate_edge_query	16
evaluate_node_query	17
evaluate_weight_spec	18
group_spatial	19
ids	20
is_sfnetwork	21
make_edges_directed	22
make_edges_explicit	23
make_edges_follow_indices	23
make_edges_implicit	24
make_edges_mixed	25
make_edges_valid	25
mozart	26
n	27
nb	27
nearest	29
nearest_ids	30
node_coordinates	31
play_spatial	32
plot.sfnetwork	33
roxel	35
sf_attr	35
sf_methods	36
sfnetwork	42
sfnetwork_to_dodgr	44
simplify_network	45
smooth_pseudo_nodes	46
spatial_centralities	47
spatial_edge_measures	48
spatial_edge_predicates	50
spatial_morphers	52

spatial_node_predicates	57
spatial_node_types	59
st_duplicated	60
st_match	61
st_network_bbox	62
st_network_blend	63
st_network_cost	65
st_network_faces	68
st_network_iso	69
st_network_join	71
st_network_paths	72
st_network_travel	76
st_project_on_network	78
st_round	80
subdivide_edges	80
tidygraph_methods	81
validate_network	82
wrap_igraph	83
Index	85

as.linnet	<i>Convert a sfnetwork into a linnet</i>
-----------	--

Description

A method to convert an object of class [sfnetwork](#) into [linnet](#) format and enhance the interoperability between sfnetworks and spatstat. Use this method without the .sfnetwork suffix and after loading the **spatstat** package.

Usage

```
as.linnet.sfnetwork(X, ...)
```

Arguments

- X An object of class [sfnetwork](#) with a projected CRS.
- ... Arguments passed to [linnet](#).

Value

An object of class [linnet](#).

See Also

[as_sfnetwork](#) to convert objects of class [linnet](#) into objects of class [sfnetwork](#).

as_s2_geography	<i>Extract the geometries of a sfnetwork as a S2 geography vector</i>
-----------------	---

Description

A method to convert an object of class `sfnetwork` into `s2_geography` format. Use this method without the `.sfnetwork` suffix and after loading the `s2` package.

Usage

```
as_s2_geography.sfnetwork(x, focused = TRUE, ...)
```

Arguments

<code>x</code>	An object of class <code>sfnetwork</code> .
<code>focused</code>	Should only features that are in focus be extracted? Defaults to TRUE. See <code>focus</code> for more information on focused networks.
<code>...</code>	Arguments passed on the corresponding <code>s2</code> function.

Value

An object of class `s2_geography`.

as_sfnetwork	<i>Convert a foreign object to a sfnetwork</i>
--------------	--

Description

Convert a given object into an object of class `sfnetwork`.

Usage

```
as_sfnetwork(x, ...)

## Default S3 method:
as_sfnetwork(x, ...)

## S3 method for class 'sf'
as_sfnetwork(x, ...)

## S3 method for class 'sfc'
as_sfnetwork(x, ...)

## S3 method for class 'dodgr_streetnet'
as_sfnetwork(x, ...)
```

```
## S3 method for class 'linnet'
as_sfnetwork(x, ...)

## S3 method for class 'psp'
as_sfnetwork(x, ...)

## S3 method for class 'sfNetwork'
as_sfnetwork(x, ...)

## S3 method for class 'tbl_graph'
as_sfnetwork(x, ...)
```

Arguments

x Object to be converted into a [sfnetwork](#).

... Additional arguments passed on to other functions.

Value

An object of class [sfnetwork](#).

Methods (by class)

- `as_sfnetwork(default)`: By default, the provided object is first converted into a [tbl_graph](#) using [as_tbl_graph](#). Further conversion into an [sfnetwork](#) will work as long as the nodes can be converted to an [sf](#) object through [st_as_sf](#). Arguments to [st_as_sf](#) can be provided as additional arguments and will be forwarded to [st_as_sf](#) through the [sfnetwork](#) construction function.
- `as_sfnetwork(sf)`: Convert spatial features of class [sf](#) directly into a [sfnetwork](#). Supported geometry types are either `LINESTRING` or `POINT`. In the first case, the lines become the edges in the network, and nodes are placed at their boundaries. Additional arguments are forwarded to [create_from_spatial_lines](#). In the latter case, the points become the nodes in the network, and are connected by edges according to a specified method. Additional arguments are forwarded to [create_from_spatial_points](#).
- `as_sfnetwork(sfc)`: Convert spatial geometries of class [sfc](#) directly into a [sfnetwork](#). Supported geometry types are either `LINESTRING` or `POINT`. In the first case, the lines become the edges in the network, and nodes are placed at their boundaries. Additional arguments are forwarded to [create_from_spatial_lines](#). In the latter case, the points become the nodes in the network, and are connected by edges according to a specified method. Additional arguments are forwarded to [create_from_spatial_points](#).
- `as_sfnetwork(dodgr_streetnet)`: Convert a directed graph of class [dodgr_streetnet](#) directly into a [sfnetwork](#). Additional arguments are forwarded to [dodgr_to_sfnetwork](#). This requires the [dodgr](#) package to be installed.
- `as_sfnetwork(linnet)`: Convert spatial linear networks of class [linnet](#) directly into a [sfnetwork](#). Additional arguments are forwarded to [create_from_spatial_lines](#). This requires the [spatstat.geom](#) package to be installed.

- `as_sfnetwork(osp)`: Convert spatial line segments of class `osp` directly into a `sfnetwork`. The lines become the edges in the network, and nodes are placed at their boundary points. Additional arguments are forwarded to `create_from_spatial_lines`.
- `as_sfnetwork(sfNetwork)`: Convert spatial networks of class `sfNetwork` from the **stplanr** package directly into a `sfnetwork`. This will extract the edges as an `sf` object and re-create the network structure. Additional arguments are forwarded to `create_from_spatial_lines`. The directness of the original network is preserved unless specified otherwise through the `directed` argument.
- `as_sfnetwork(tbl_graph)`: Convert graph objects of class `tbl_graph` directly into a `sfnetwork`. This will work if at least the nodes can be converted to an `sf` object through `st_as_sf`. Arguments to `st_as_sf` can be provided as additional arguments and will be forwarded to `st_as_sf` through the `sfnetwork` construction function. The directness of the original graph is preserved unless specified otherwise through the `directed` argument.

Examples

```
# From an sf object with LINESTRING geometries.
library(sf, quietly = TRUE)

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1), mfrow = c(1,2))

as_sfnetwork(roxel)

plot(st_geometry(roxel))
plot(as_sfnetwork(roxel))

# From an sf object with POINT geometries.
# For more examples see ?create_from_spatial_points.
as_sfnetwork(mozart)

plot(st_geometry(mozart))
plot(as_sfnetwork(mozart))

par(oldpar)

# From a dodgr_streetnet object.
if (require(dodgr, quietly = TRUE) & require(geodist, quietly = TRUE)) {
  as_sfnetwork(dodgr::weight_streetnet(hampi))
}

# From a linnet object.
if (require(spatstat.geom, quietly = TRUE)) {
  as_sfnetwork(simplenet)
}

# From a osp object.
if (require(spatstat.geom, quietly = TRUE)) {
  set.seed(42)
  test_osp = osp(runif(10), runif(10), runif(10), runif(10), window=owin())
  as_sfnetwork(test_osp)
```

```

}

# From a tbl_graph with coordinate columns.
library(tidygraph, quietly = TRUE)

nodes = data.frame(lat = c(7, 7, 8), lon = c(51, 52, 52))
edges = data.frame(from = c(1, 1, 3), to = c(2, 3, 2))
tbl_net = tbl_graph(nodes, edges)
as_sfnetwork(tbl_net, coords = c("lon", "lat"), crs = 4326)

```

as_tibble

Extract the active element of a sfnetwork as spatial tibble

Description

The `sfnetwork` method for `as_tibble` is conceptually different. Whenever a geometry list column is present, it will by default return what we call a 'spatial tibble'. With that we mean an object of class `c('sf', 'tbl_df')` instead of an object of class `'tbl_df'`. This little conceptual trick is essential for how tidyverse functions handle `sfnetwork` objects, i.e. always using the corresponding `sf` method if present. When using `as_tibble` on `sfnetwork` objects directly as a user, you can disable this behaviour by setting `spatial = FALSE`.

Usage

```

## S3 method for class 'sfnetwork'
as_tibble(x, active = NULL, focused = TRUE, spatial = TRUE, ...)

```

Arguments

<code>x</code>	An object of class <code>sfnetwork</code> .
<code>active</code>	Which network element (i.e. nodes or edges) to activate before extracting. If <code>NULL</code> , it will be set to the current active element of the given network. Defaults to <code>NULL</code> .
<code>focused</code>	Should only features that are in focus be extracted? Defaults to <code>TRUE</code> . See <code>focus</code> for more information on focused networks.
<code>spatial</code>	Should the extracted tibble be a 'spatial tibble', i.e. an object of class <code>c('sf', 'tbl_df')</code> , if it contains a geometry list column. Defaults to <code>TRUE</code> .
<code>...</code>	Arguments passed on to <code>as_tibble</code> .

Value

The active element of the network as an object of class `sf` if a geometry list column is present and `spatial = TRUE`, and object of class `tibble` otherwise.

Examples

```
library(tibble, quietly = TRUE)

net = as_sfnetwork(roxel)

# Extract the active network element as a spatial tibble.
as_tibble(net)

# Extract any network element as a spatial tibble.
as_tibble(net, "edges")

# Extract the active network element as a regular tibble.
as_tibble(net, spatial = FALSE)
```

autoplot

Plot sfnetwork geometries with ggplot2

Description

Plot the geometries of an object of class `sfnetwork` automatically as a `ggplot` object. Use this method without the `.sfnetwork` suffix and after loading the `ggplot2` package.

Usage

```
autoplot.sfnetwork(object, ...)
```

Arguments

<code>object</code>	An object of class <code>sfnetwork</code> .
<code>...</code>	Ignored.

Details

See `autoplot`.

Value

An object of class `ggplot`.

bind_spatial	<i>Add nodes or edges to a spatial network.</i>
--------------	---

Description

These functions are the spatially aware versions of tidygraph's [bind_nodes](#) and [bind_edges](#) that allow you to add rows to the nodes or edges tables in a [sfnetwork](#) object. As with [bind_rows](#) columns are matched by name and filled with NA if the column does not exist in some instances.

Usage

```
bind_spatial_nodes(.data, ...)
```

```
bind_spatial_edges(.data, ..., node_key = "name", force = FALSE)
```

Arguments

.data	An object of class sfnetwork .
...	One or more objects of class sf containing the nodes or edges to be added.
node_key	The name of the column in the nodes table that character represented to and from columns should be matched against. If NA, the first column is always chosen. This setting has no effect if to and from are given as integers. Defaults to 'name'.
force	Should network validity checks be skipped? Defaults to FALSE, meaning that network validity checks are executed after binding edges, making sure that boundary points of edges match their corresponding node coordinates.

Value

An object of class [sfnetwork](#) with added nodes or edges.

Examples

```
library(sf, quietly = TRUE)
library(dplyr, quietly = TRUE)

net = roxel |>
  slice(c(1:2)) |>
  st_transform(3035) |>
  as_sfnetwork()

pts = roxel |>
  slice(c(3:4)) |>
  st_transform(3035) |>
  st_centroid()

bind_spatial_nodes(net, pts)
```

contract_nodes

*Contract groups of nodes in a spatial network***Description**

Combine groups of nodes into a single node per group. The centroid such a group will be used by default as new geometry of the contracted node. If edges are spatially explicit, edge geometries are updated accordingly such that the valid spatial network structure is preserved.

Usage

```
contract_nodes(
  x,
  groups,
  simplify = TRUE,
  compute_centroids = TRUE,
  reconnect_edges = TRUE,
  attribute_summary = "ignore",
  store_original_ids = FALSE,
  store_original_data = FALSE
)
```

Arguments

x	An object of class sfnetwork .
groups	A group index for each node in x.
simplify	Should the network be simplified after contraction? Defaults to TRUE. This means that multiple edges and loop edges will be removed. Multiple edges are introduced by contraction when there are several connections between the same groups of nodes. Loop edges are introduced by contraction when there are connections within a group. Note however that setting this to TRUE also removes multiple edges and loop edges that already existed before contraction.
compute_centroids	Should the new geometry of each contracted group of nodes be the centroid of all group members? Defaults to TRUE. If set to FALSE, the geometry of the first node in each group will be used instead, which requires considerably less computing time.
reconnect_edges	Should the geometries of the edges be updated such they match the new node geometries? Defaults to TRUE. Only set this to FALSE if you know the node geometries did not change, otherwise the valid spatial network structure is broken.
attribute_summary	How should the attributes of contracted nodes be summarized? There are several options, see igraph-attribute-combination for details.

store_original_ids

For each group of contracted nodes, should the indices of the original nodes be stored as an attribute of the new edge, in a column named `.tidygraph_node_index`? This is in line with the design principles of tidygraph. Defaults to FALSE.

store_original_data

For each group of contracted nodes, should the data of the original nodes be stored as an attribute of the new edge, in a column named `.orig_data`? This is in line with the design principles of tidygraph. Defaults to FALSE.

Value

The contracted network as object of class `sfnetwork`.

create_from_spatial_lines

Create a spatial network from linestring geometries

Description

Create a spatial network from linestring geometries

Usage

```
create_from_spatial_lines(
  x,
  directed = TRUE,
  compute_length = FALSE,
  subdivide = FALSE
)
```

Arguments

- | | |
|-----------------------|---|
| x | An object of class <code>sf</code> or <code>sfc</code> with LINESTRING geometries. |
| directed | Should the constructed network be directed? Defaults to TRUE. |
| compute_length | Should the geographic length of the edges be stored in a column named <code>length</code> ? Uses <code>st_length</code> to compute the length of the edge geometries. If there is already a column named <code>length</code> , it will be overwritten. Please note that the values in this column are not automatically recognized as edge weights. This needs to be specified explicitly when calling a function that uses edge weights. Defaults to FALSE. |
| subdivide | Should the given linestring geometries be subdivided at locations where an interior point is equal to an interior or boundary point in another feature? This will connect the features at those locations. Defaults to FALSE, meaning that features are only connected at their boundaries. |

Details

It is assumed that the given linestring geometries form the edges in the network. Nodes are created at the line boundaries. Shared boundaries between multiple linestrings become the same node.

Value

An object of class `sfnetwork`.

Note

By default `sfnetworks` rounds coordinates to 12 decimal places to determine spatial equality. You can influence this behavior by explicitly setting the precision of the linestrings using `st_set_precision`.

See Also

`create_from_spatial_points`

Examples

```
library(sf, quietly = TRUE)

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1), mfrow = c(1,2))

net = as_sfnetwork(roxel)
net

plot(st_geometry(roxel))
plot(net)

par(oldpar)
```

`create_from_spatial_points`*Create a spatial network from point geometries*

Description

Create a spatial network from point geometries

Usage

```
create_from_spatial_points(
  x,
  connections = "complete",
  directed = TRUE,
  edges_as_lines = TRUE,
```

```

    compute_length = FALSE,
    k = 1
)
```

Arguments

x	An object of class sf or sfc with POINT geometries.
connections	How to connect the given point geometries to each other? Can be specified either as an adjacency matrix, or as a character describing a specific method to define the connections. See Details.
directed	Should the constructed network be directed? Defaults to TRUE.
edges_as_lines	Should the created edges be spatially explicit, i.e. have LINESTRING geometries stored in a geometry list column? Defaults to TRUE.
compute_length	Should the geographic length of the edges be stored in a column named length? Uses st_length to compute the length of the edge geometries when edges are spatially explicit, and st_distance to compute the distance between boundary nodes when edges are spatially implicit. Please note that the values in this column are not automatically recognized as edge weights. This needs to be specified explicitly when calling a function that uses edge weights. Defaults to FALSE.
k	The amount of neighbors to connect to if connections = 'knn'. Defaults to 1, meaning that nodes are only connected to their nearest neighbor. Ignored for any other value of the connections argument.

Details

It is assumed that the given points form the nodes in the network. How those nodes are connected by edges depends on the connections argument.

The connections can be specified through an adjacency matrix A, which is an n x n matrix with n being the number of nodes, and element A_{ij} holding a TRUE value if there is an edge from node i to node j, and a FALSE value otherwise. In the case of undirected networks, the matrix is not tested for symmetry, and an edge will exist between node i and node j if either element A_{ij} or element A_{ji} is TRUE. Non-logical matrices are first converted into logical matrices using [as.logical](#), whenever possible.

The provided adjacency matrix may also be sparse. This can be an object of one of the sparse matrix classes from the **Matrix** package, or a list-formatted sparse matrix. This is a list with one element per node, holding the integer indices of the nodes it is adjacent to. An example are [sgbp](#) objects. If the values are not integers, they are first converted into integers using [as.integer](#), whenever possible.

Alternatively, the connections can be specified by providing the name of a specific method that will create the adjacency matrix internally. Valid options are:

- complete: All nodes are directly connected to each other.
- sequence: The nodes are sequentially connected to each other, meaning that the first node is connected to the second node, the second node is connected to the third node, et cetera.

- **mst**: The nodes are connected by their spatial **minimum spanning tree**, i.e. the set of edges with the minimum total edge length required to connect all nodes. The tree is always constructed on an undirected network, regardless of the value of the `directed` argument. If `directed = TRUE`, each edge is duplicated and reversed to ensure full connectivity of the network. Can also be specified as `minimum_spanning_tree`.
- **delaunay**: The nodes are connected by their **Delaunay triangulation**. Requires the **spdep** package to be installed, and assumes planar coordinates.
- **gabriel**: The nodes are connected as a **Gabriel graph**. Requires the **spdep** package to be installed, and assumes planar coordinates.
- **rn**: The nodes are connected as a **relative neighborhood graph**. Can also be specified as `relative_neighborhood` or `relative_neighbourhood`. Requires the **spdep** package to be installed, and assumes planar coordinates.
- **knn**: Each node is connected to its `k` nearest neighbors, with `k` being specified through the `k` argument. By default, `k = 1`, meaning that the nodes are connected as a **nearest neighbor graph**. Can also be specified as `nearest_neighbors` or `nearest_neighbours`. Requires the **spdep** package to be installed.

Value

An object of class `sfnetwork`.

See Also

`create_from_spatial_lines`, `play_geometric`

Examples

```
library(sf, quietly = TRUE)

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1))

pts = st_transform(mozart, 3035)

# Using a custom adjacency matrix
adj = matrix(c(rep(TRUE, 17), rep(rep(FALSE, 17), 16)), nrow = 17)
net = as_sfnetwork(pts, connections = adj)

plot(net)

# Using a sparse adjacency matrix from a spatial predicate
dst = units::set_units(300, "m")
adj = st_is_within_distance(pts, dist = dst, remove_self = TRUE)
net = as_sfnetwork(pts, connections = adj, directed = FALSE)

plot(net)

# Using pre-defined methods
cnet = as_sfnetwork(pts, connections = "complete")
snet = as_sfnetwork(pts, connections = "sequence")
```

```

mnet = as_sfnetwork(pts, connections = "mst")
dnet = as_sfnetwork(pts, connections = "delaunay")
gnet = as_sfnetwork(pts, connections = "gabriel")
rnet = as_sfnetwork(pts, connections = "rn")
nnet = as_sfnetwork(pts, connections = "knn")
knet = as_sfnetwork(pts, connections = "knn", k = 2)

par(mar = c(1,1,1,1), mfrow = c(4,2))

plot(cnet, main = "complete")
plot(snet, main = "sequence")
plot(mnet, main = "minimum spanning tree")
plot(dnet, main = "delaunay triangulation")
plot(gnet, main = "gabriel graph")
plot(rnet, main = "relative neighborhood graph")
plot(nnet, main = "nearest neighbor graph")
plot(knet, main = "k nearest neighbor graph (k = 2)")

par(oldpar)

```

data

Extract the node or edge data from a spatial network

Description

Extract the node or edge data from a spatial network

Usage

```

node_data(x, focused = TRUE)

edge_data(x, focused = TRUE, require_sf = FALSE)

```

Arguments

x	An object of class sfnetwork .
focused	Should only features that are in focus be extracted? Defaults to TRUE. See focus for more information on focused networks.
require_sf	Is an sf object required? This will make extraction of edge data fail if the edges are spatially implicit. Defaults to FALSE.

Value

For the nodes, always an object of class [sf](#). For the edges, an object of class [sf](#) if the edges are spatially explicit, and an object of class [tibble](#) if the edges are spatially implicitly and `require_sf = FALSE`.

Examples

```
net = as_sfnetwork(roxel[1:10, ])
node_data(net)
edge_data(net)
```

dual_weights	<i>Specify dual edge weights</i>
--------------	----------------------------------

Description

Dual edge weights are two sets of edge weights, one (the actual weight) to determine the shortest path, and the other (the reported weight) to report the cost of that path.

Usage

```
dual_weights(reported, actual)
```

Arguments

- reported The edge weights to be reported. Evaluated by [evaluate_weight_spec](#).
- actual The actual edge weights to be used to determine shortest paths. Evaluated by [evaluate_weight_spec](#).

Details

Dual edge weights enable dual-weighted routing. This is supported by the [dodgr](#) routing backend.

Value

An object of class dual_weights.

evaluate_edge_query	<i>Query specific edge indices from a spatial network</i>
---------------------	---

Description

This function is not meant to be called directly, but used inside other functions that accept an edge query.

Usage

```
evaluate_edge_query(data, query)
```

Arguments

data	An object of class sfnetwork .
query	The query that defines for which edges to extract indices, defused into a quosure . See Details for the different ways in which edge queries can be formulated.

Details

There are multiple ways in which edge indices can be queried in sfnetworks. The query can be formatted as follows:

- As spatial features: Spatial features can be given as object of class [sf](#) or [sfc](#). The nearest edge to each feature is found by calling [st_nearest_feature](#).
- As edge type query function: A [edge type query function](#) defines for each edge if it is of a given type or not. Nodes that meet the criterium are queried.
- As edge predicate query function: A [edge predicate query function](#) defines for each edge if a given spatial predicate applies to the spatial relation between that edge and other spatial features. Nodes that meet the criterium are queried.
- As column name: The referenced column is expected to have logical values defining for each edge if it should be queried or not. Note that tidy evaluation is used and hence the column name should be unquoted.
- As integers: Integers are interpreted as edge indices. A edge index corresponds to a row-number in the edges table of the network.
- As characters: Characters are interpreted as edge names. A edge name corresponds to a value in a column named "name" in the the edges table of the network. Note that this column is expected to store unique names without any duplicated values.
- As logicals: Logicals should define for each edge if it should be queried or not.

Queries that can not be evaluated in any of the ways described above will be forcefully converted to integers using [as.integer](#).

Value

A vector of queried edge indices.

evaluate_node_query	<i>Query specific node indices from a spatial network</i>
---------------------	---

Description

This function is not meant to be called directly, but used inside other functions that accept a node query.

Usage

```
evaluate_node_query(data, query)
```

Arguments

data	An object of class <code>sfnetwork</code> .
query	The query that defines for which nodes to extract indices, defused into a <code>quosure</code> . See Details for the different ways in which node queries can be formulated.

Details

There are multiple ways in which node indices can be queried in `sfnetworks`. The query can be formatted as follows:

- As spatial features: Spatial features can be given as object of class `sf` or `sfc`. The nearest node to each feature is found by calling `st_nearest_feature`.
- As node type query function: A `node type query function` defines for each node if it is of a given type or not. Nodes that meet the criterium are queried.
- As node predicate query function: A `node predicate query function` defines for each node if a given spatial predicate applies to the spatial relation between that node and other spatial features. Nodes that meet the criterium are queried.
- As column name: The referenced column is expected to have logical values defining for each node if it should be queried or not. Note that tidy evaluation is used and hence the column name should be unquoted.
- As integers: Integers are interpreted as node indices. A node index corresponds to a row-number in the nodes table of the network.
- As characters: Characters are interpreted as node names. A node name corresponds to a value in a column named "name" in the the nodes table of the network. Note that this column is expected to store unique names without any duplicated values.
- As logicals: Logicals should define for each node if it should be queried or not.

Queries that can not be evaluated in any of the ways described above will be forcefully converted to integers using `as.integer`.

Value

A vector of queried node indices.

evaluate_weight_spec	<i>Specify edge weights in a spatial network</i>
----------------------	--

Description

This function is not meant to be called directly, but used inside other functions that accept the specification of edge weights.

Usage

```
evaluate_weight_spec(data, spec)
```

Arguments

data	An object of class sfnetwork .
spec	The specification that defines how to compute or extract edge weights defused into a quosure . See Details for the different ways in which edge weights can be specified.

Details

There are multiple ways in which edge weights can be specified in sfnetworks. The specification can be formatted as follows:

- As edge measure function: A [spatial edge measure function](#) computes a given measure for each edge, which will then be used as edge weights.
- As column name: A column in the edges table of the network that contains the edge weights. Note that tidy evaluation is used and hence the column name should be unquoted.
- As a numeric vector: This vector should be of the same length as the number of edges in the network, specifying for each edge what its weight is.
- As dual weights: Dual weights can be specified by the [dual_weights](#) function. This allows to use a different set of weights for shortest paths computation and for reporting the total cost of those paths. Note that not every routing backend support dual-weighted routing.

If the weight specification is NULL or NA, this means that no edge weights are used. For shortest path computation, this means that the shortest path is simply the path with the fewest number of edges.

Value

A numeric vector of edge weights.

Note

For backward compatibility it is currently also still possible to format the specification as a quoted column name, but this may be removed in future versions.

Also note that many shortest path algorithms require edge weights to be positive.

group_spatial	<i>Group nodes based on spatial distance</i>
---------------	--

Description

These functions forms a spatial extension to the [grouping](#) functions in tidygraph, allowing to detect communities with spatial clustering algorithms.

Usage

```
group_spatial_dbscan(epsilon, min_pts = 1, use_network_distance = TRUE, ...)
```

Arguments

<code>epsilon</code>	The value of the epsilon parameter for the DBSCAN clustering algorithm, defining the radius of the neighborhood of a node.
<code>min_pts</code>	The value of the minPts parameter for the DBSCAN clustering algorithm, defining the minimum number of points in the neighborhood to be considered a core point.
<code>use_network_distance</code>	Should the distance between nodes be computed as the distance over the network (using <code>st_network_distance</code>)? Defaults to TRUE. If set to FALSE, the straight-line distance (using <code>st_distance</code>) is computed instead.
<code>...</code>	Additional arguments passed on to the clustering algorithm.

Details

Just as with all grouping functions in tidygraph, spatial grouping functions are meant to be called inside tidygraph verbs such as `mutate` or `filter`, where the network that is currently being worked on is known and thus not needed as an argument to the function. If you want to use an algorithm outside of the tidygraph framework you can use `with_graph` to set the context temporarily while the algorithm is being evaluated.

Value

A numeric vector with the membership for each node in the network. The enumeration happens in order based on group size progressing from the largest to the smallest group.

Functions

- `group_spatial_dbscan()`: Uses density-based spatial clustering as implemented in the `dbscan` function of the `dbscan` package. This requires the `dbscan` package to be installed. Each node marked as noise will form its own cluster.

Examples

```
library(tidygraph, quietly = TRUE)

play_geometric(10, 0.5) |>
  activate(nodes) |>
  mutate(group = group_spatial_dbscan(0.25))
```

ids

Extract all node or edge indices from a spatial network

Description

Extract all node or edge indices from a spatial network

Usage

```
node_ids(x, focused = TRUE)
```

```
edge_ids(x, focused = TRUE)
```

Arguments

x An object of class `sfnetwork`.

focused Should only the indices of features that are in focus be extracted? Defaults to TRUE. See [focus](#) for more information on focused networks.

Details

The indices in these objects are always integers that correspond to rownumbers in respectively the nodes or edges table.

Value

A vector of integers.

Examples

```
net = as_sfnetwork(roxel[1:10, ])
node_ids(net)
edge_ids(net)
```

is_sfnetwork	<i>Check if an object is a sfnetwork</i>
--------------	--

Description

Check if an object is a sfnetwork

Usage

```
is_sfnetwork(x)
```

```
is.sfnetwork(x)
```

Arguments

x Object to be checked.

Value

TRUE if the given object is an object of class `sfnetwork`, FALSE otherwise.

Examples

```
library(tidygraph, quietly = TRUE, warn.conflicts = FALSE)

net = as_sfnetwork(roxel)
is_sfnetwork(net)
is_sfnetwork(as_tbl_graph(net))
```

make_edges_directed	<i>Convert undirected edges into directed edges based on their geometries</i>
---------------------	---

Description

This function converts an undirected network to a directed network following the direction given by the linestring geometries of the edges.

Usage

```
make_edges_directed(x)
```

Arguments

x An object of class `sfnetwork`.

Details

In undirected spatial networks it is required that the boundary of edge geometries contain their incident node geometries. However, it is not required that their start point equals their specified `*from*` node and their end point their specified `*to*` node. Instead, it may be vice versa. This is because for undirected networks `*from*` and `*to*` indices are always swopped if the `*to*` index is lower than the `*from*` index. Therefore, the direction given by the `*from*` and `*to*` indices does not necessarily match the direction given by the edge geometries.

Value

A directed network as object of class `sfnetwork`.

Note

If the network is already directed it is returned unmodified.

make_edges_explicit	<i>Construct edge geometries for spatially implicit networks</i>
---------------------	--

Description

This function turns spatially implicit networks into spatially explicit networks by adding a geometry column to the edge data.

Usage

```
make_edges_explicit(x, ...)
```

Arguments

x	An object of class <code>sfnetwork</code> .
...	Arguments forwarded to <code>st_as_sf</code> to directly convert the edges table into a <code>sf</code> object. If no arguments are given, the edges are made explicit by simply drawing straight lines between the start and end node of each edge.

Value

An object of class `sfnetwork` with spatially explicit edges.

Note

If the network is already spatially explicit it is returned unmodified.

make_edges_follow_indices	<i>Match the direction of edge geometries to their specified incident nodes</i>
---------------------------	---

Description

This function updates edge geometries in undirected networks such that they are guaranteed to start at their specified `*from*` node and end at their specified `*to*` node.

Usage

```
make_edges_follow_indices(x)
```

Arguments

x	An object of class <code>sfnetwork</code> .
---	---

Details

In undirected spatial networks it is required that the boundary of edge geometries contain their incident node geometries. However, it is not required that their start point equals their specified **from** node and their end point their specified **to** node. Instead, it may be vice versa. This is because for undirected networks **from** and **to** indices are always swapped if the **to** index is lower than the **from** index.

This function reverses edge geometries if they start at the **to** node and end at the **from** node, such that in the resulting network it is guaranteed that edge boundary points exactly match their incident node geometries. In directed networks, there will be no change.

Value

An object of class `sfnetwork` with updated edge geometries.

make_edges_implicit	<i>Drop edge geometries of spatially explicit networks</i>
---------------------	--

Description

This function turns spatially explicit networks into spatially implicit networks by dropping the geometry column of the edge data.

Usage

```
make_edges_implicit(x)
```

Arguments

`x` An object of class `sfnetwork`.

Value

An object of class `sfnetwork` with spatially implicit edges.

Note

If the network is already spatially implicit it is returned unmodified.

make_edges_mixed	<i>Make some edges directed and some undirected</i>
------------------	---

Description

This function creates a mixed network, meaning that some edges are directed, and some are undirected. In practice this is implemented as a directed network in which those edges that are meant to be undirected are duplicated and reversed.

Usage

```
make_edges_mixed(x, directed)
```

Arguments

x	An object of class sfnetwork .
directed	An integer vector of edge indices specifying those edges that should be directed.

Value

A mixed network as object of class [sfnetwork](#).

make_edges_valid	<i>Match edge geometries to their incident node locations</i>
------------------	---

Description

This function makes invalid edges valid by modifying either edge or node geometries such that the boundary points of edge linestring geometries always match the point geometries of the nodes that are specified as their incident nodes by the **from** and **to** columns.

Usage

```
make_edges_valid(x, preserve_geometries = FALSE)
```

Arguments

x	An object of class sfnetwork .
preserve_geometries	Should the edge geometries remain unmodified? Defaults to FALSE. See Details.

Details

If geometries should be preserved, edges are made valid by adding edge boundary points that do not equal their corresponding node geometry as new nodes to the network, and updating the **from** and **to** indices to match this newly added nodes. If FALSE, edges are made valid by modifying their geometries, i.e. edge boundary points that do not equal their corresponding node geometry are replaced by that node geometry.

Value

An object of class `sfnetwork` with corrected edge geometries.

Note

This function works only if the edge geometries are meant to start at their specified **from** node and end at their specified **to** node. In undirected networks this is not necessarily the case, since edge geometries are allowed to start at their specified **to** node and end at their specified **from** node. Therefore, in undirected networks those edges first have to be reversed before running this function. Use `make_edges_follow_indices` for this.

mozart

Point locations for places about W. A. Mozart in Salzburg, Austria

Description

A dataset containing point locations (museums, sculptures, squares, universities, etc.) of places named after Wolfgang Amadeus Mozart in the city of Salzburg, Austria. The data are taken from OpenStreetMap. See 'data-raw/mozart.R' for code on its creation.

Usage

```
mozart
```

Format

An object of class `sf` with LINESTRING geometries, containing 17 features and four columns:

name the name of the point location

type the type of location, e.g. museum, artwork, cinema, etc.

website the website URL for more information about the place, if available

geometry the geometry list column

Source

<https://www.openstreetmap.org>

n	<i>Count the number of nodes or edges in a network</i>
---	--

Description

Count the number of nodes or edges in a network

Usage

```
n_nodes(x, focused = FALSE)
```

```
n_edges(x, focused = FALSE)
```

Arguments

x	An object of class sfnetwork , or any other network object inheriting from igraph .
focused	Should only features that are in focus be counted? Defaults to FALSE. See focus for more information on focused networks.

Value

An integer.

Examples

```
net = as_sfnetwork(roxel)
n_nodes(net)
n_edges(net)
```

nb	<i>Conversion between neighbor lists and sfnetworks</i>
----	---

Description

Neighbor lists are sparse adjacency matrices in list format that specify for each node to which other nodes it is adjacent. They occur for example in the **sf** package as [sgbp](#) objects, and are also frequently used in the **spdep** package.

Usage

```
nb_to_sfnetwork(
  x,
  nodes,
  directed = TRUE,
  edges_as_lines = TRUE,
  compute_length = FALSE,
  force = FALSE
)

sfnetwork_to_nb(x, direction = "out")
```

Arguments

x	For the conversion to sfnetwork: a neighbor list, which is a list adjacent to. For the conversion from sfnetwork: an object of class sfnetwork .
nodes	The nodes themselves as an object of class sf or sfc with POINT geometries.
directed	Should the constructed network be directed? Defaults to TRUE.
edges_as_lines	Should the created edges be spatially explicit, i.e. have LINESTRING geometries stored in a geometry list column? Defaults to TRUE.
compute_length	Should the geographic length of the edges be stored in a column named length? Defaults to FALSE.
force	Should validity checks be skipped? Defaults to FALSE, meaning that network validity checks are executed when constructing the network. These checks make sure that the provided neighbor list has a valid structure, i.e. that its length is equal to the number of provided nodes and that its values are all integers referring to one of the nodes.
direction	The direction that defines if two nodes are neighbors. Defaults to 'out', meaning that the direction given by the network is followed and node j is only a neighbor of node i if there exists an edge i->j. May be set to 'in', meaning that the opposite direction is followed and node j is only a neighbor of node i if there exists an edge j->i. May also be set to 'all', meaning that the network is considered to be undirected. This argument is ignored for undirected networks.

Value

For the conversion to sfnetwork: An object of class [sfnetwork](#). For the conversion from sfnetwork: a neighbor list, which is a list with one element per node that holds the integer indices of the nodes it is adjacent to.

nearest	<i>Extract the nearest nodes or edges to given spatial features</i>
---------	---

Description

Extract the nearest nodes or edges to given spatial features

Usage

```
nearest_nodes(x, y, focused = TRUE)
```

```
nearest_edges(x, y, focused = TRUE)
```

Arguments

x	An object of class sfnetwork .
y	Spatial features as object of class sf or sfc .
focused	Should only features that are in focus be extracted? Defaults to TRUE. See focus for more information on focused networks.

Details

To determine the nearest node or edge to each feature in y the function [st_nearest_feature](#) is used. When extracting nearest edges, spatially explicit edges are required, i.e. the edges table should have a geometry column.

Value

An object of class [sf](#) with each row containing the nearest node or edge to the corresponding spatial features in y.

Examples

```
library(sf, quietly = TRUE)

net = as_sfnetwork(roxel)
pts = st_sample(st_bbox(roxel), 6)

nodes = nearest_nodes(net, pts)
edges = nearest_edges(net, pts)

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1), mfrow = c(1,2))

plot(net, main = "Nearest nodes")
plot(pts, cex = 2, col = "red", pch = 20, add = TRUE)
plot(st_geometry(nodes), cex = 2, col = "orange", pch = 20, add = TRUE)
```

```

plot(net, main = "Nearest edges")
plot(pts, cex = 2, col = "red", pch = 20, add = TRUE)
plot(st_geometry(edges), lwd = 2, col = "orange", pch = 20, add = TRUE)

par(oldpar)

```

nearest_ids	<i>Extract the indices of nearest nodes or edges to given spatial features</i>
-------------	--

Description

Extract the indices of nearest nodes or edges to given spatial features

Usage

```

nearest_node_ids(x, y, focused = TRUE)

nearest_edge_ids(x, y, focused = TRUE)

```

Arguments

x	An object of class sfnetwork .
y	Spatial features as object of class sf or sfc .
focused	Should only the indices of features that are in focus be extracted? Defaults to TRUE. See focus for more information on focused networks.

Details

To determine the nearest node or edge to each feature in y the function [st_nearest_feature](#) is used. When extracting nearest edges, spatially explicit edges are required, i.e. the edges table should have a geometry column.

Value

An integer vector with each element containing the index of the nearest node or edge to the corresponding spatial features in y.

Examples

```

library(sf, quietly = TRUE)

net = as_sfnetwork(roxel)
pts = st_sample(st_bbox(roxel), 6)

nearest_node_ids(net, pts)
nearest_edge_ids(net, pts)

```

node_coordinates	<i>Query node coordinates</i>
------------------	-------------------------------

Description

These functions allow to query specific coordinate values from the geometries of the nodes.

Usage

node_X()

node_Y()

node_Z()

node_M()

Details

Just as with all query functions in tidygraph, these functions are meant to be called inside tidygraph verbs such as [mutate](#) or [filter](#), where the network that is currently being worked on is known and thus not needed as an argument to the function. If you want to use an algorithm outside of the tidygraph framework you can use [with_graph](#) to set the context temporarily while the algorithm is being evaluated.

Value

A numeric vector of the same length as the number of nodes in the network.

Note

If a requested coordinate value is not available for a node, NA will be returned.

Examples

```
library(sf, quietly = TRUE)
library(tidygraph, quietly = TRUE)

# Create a network.
net = as_sfnetwork(roxel)

# Use query function in a filter call.
filtered = net |>
  activate(nodes) |>
  filter(node_X() > 7.54)

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1))
plot(net, col = "grey")
```

```

plot(filtered, col = "red", add = TRUE)
par(oldpar)

# Use query function in a mutate call.
net |>
  activate(nodes) |>
  mutate(X = node_X(), Y = node_Y())

# Use query function directly.
X = with_graph(net, node_X())
head(X)

```

play_spatial

Create random spatial networks

Description

Random spatial networks are created by randomly sampling nodes within a given area, and connecting them by edges according to a specified method.

Usage

```

play_geometric(
  n,
  radius,
  bounds = NULL,
  edges_as_lines = TRUE,
  compute_length = FALSE,
  ...
)

```

Arguments

n	The number of nodes to be sampled.
radius	The radius within which nodes will be connected by an edge.
bounds	The spatial features within which the nodes should be sampled as object of class sf , sfc , sfg or bbox . If set to NULL, nodes will be sampled within a unit square.
edges_as_lines	Should the created edges be spatially explicit, i.e. have LINESTRING geometries stored in a geometry list column? Defaults to TRUE.
compute_length	Should the geographic length of the edges be stored in a column named length? Uses st_length to compute the length of the edge geometries when edges are spatially explicit, and st_distance to compute the distance between boundary nodes when edges are spatially implicit. Please note that the values in this column are not automatically recognized as edge weights. This needs to be specified explicitly when calling a function that uses edge weights. Defaults to FALSE.
...	Additional arguments passed on to st_sample . Ignored if bounds = NULL.

Functions

- `play_geometric()`: Creates a random geometric graph. Two nodes will be connected by an edge if the distance between them is within the given radius. If nodes are sampled on a unit square (i.e. when `bounds = NULL`) this radius is unitless. If bounds are given as a spatial feature, the radius is assumed to be in meters for geographic coordinates, and in the units of the coordinate reference system for projected coordinates. Alternatively, units can also be specified explicitly by providing a `units` object.

Examples

```
library(sf, quietly = TRUE)

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1))

# Sample 10 nodes on a unit square
# Connect nodes by an edge if they are within 0.25 distance from each other
net = play_geometric(10, 0.25)
net

plot(net)

# Sample 10 nodes within a spatial bounding box
# Connect nodes by an edge if they are within 1 km from each other
net = play_geometric(10, units::set_units(1, "km"), bounds = st_bbox(roxel))
net

plot(net)

par(oldpar)
```

plot.sfnetwork

Plot the geometries of a sfnetwork

Description

Plot the geometries of an object of class `sfnetwork`.

Usage

```
## S3 method for class 'sfnetwork'
plot(x, draw_lines = TRUE, node_args = list(), edge_args = list(), ...)
```

Arguments

`x` Object of class `sfnetwork`.

<code>draw_lines</code>	If the edges of the network are spatially implicit, should straight lines be drawn between connected nodes? Defaults to TRUE. Ignored when the edges of the network are spatially explicit.
<code>node_args</code>	A named list of arguments that will be passed on to <code>plot.sf</code> only for plotting the nodes.
<code>edge_args</code>	A named list of arguments that will be passed on to <code>plot.sf</code> only for plotting the edges.
<code>...</code>	Arguments passed on to <code>plot.sf</code> that will apply to the plot as a whole.

Details

Arguments passed to `...` will be used both for plotting the nodes and for plotting the edges. Edges are always plotted first. Arguments specified in `node_args` and `edge_args` should not be specified in `...` as well, this will result in an error.

Value

Invisible.

Examples

```
library(sf, quietly = TRUE)

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1), mfrow = c(1,1))
net = as_sfnetwork(roxel)
plot(net)

# When edges are spatially implicit.
# By default straight lines will be drawn between connected nodes.
par(mar = c(1,1,1,1), mfrow = c(1,2))
inet = st_drop_geometry(activate(net, "edges"))
plot(inet)
plot(inet, draw_lines = FALSE)

# Changing plot settings.
par(mar = c(1,1,1,1), mfrow = c(1,1))
plot(net, main = "My network", col = "blue", pch = 18, lwd = 1, cex = 2)

# Changing plot settings for nodes and edges separately.
plot(net, node_args = list(col = "red"), edge_args = list(col = "blue"))

# Add grid and axis
par(mar = c(2.5,2.5,1,1))
plot(net, graticule = TRUE, axes = TRUE)

# Plot two networks on top of each other.
par(mar = c(1,1,1,1), mfrow = c(1,1))
neta = as_sfnetwork(roxel[1:10, ])
netb = as_sfnetwork(roxel[50:60, ])
plot(neta)
```

```
plot(netb, node_args = list(col = "orange"), add = TRUE)

par(oldpar)
```

roxel

Road network of Münster Roxel

Description

A dataset containing the road network (roads, bikelanes, footpaths, etc.) of Roxel, a neighborhood in the city of Münster, Germany. The data are taken from OpenStreetMap, querying by key = 'highway'. See 'data-raw/roxel.R' for code on its creation.

Usage

```
roxel
```

Format

An object of class `sf` with LINESTRING geometries, containing 1215 features and three columns:

name the name of the road, if it exists

type the type of the road, e.g. cycleway

geometry the geometry list column

Source

<https://www.openstreetmap.org>

sf_attr

Query sf attributes from the active element of a sfnetwork

Description

Query sf attributes from the active element of a sfnetwork

Usage

```
sf_attr(x, name, active = NULL)
```

Arguments

<code>x</code>	An object of class <code>sfnetwork</code> .
<code>name</code>	Name of the attribute to query. Either 'sf_column' to extract the name of the geometry list column, or 'agr' to extract the specification of attribute-geometry relationships.
<code>active</code>	Which network element (i.e. nodes or edges) to activate before extracting. If NULL, it will be set to the current active element of the given network. Defaults to NULL.

Value

The value of the queried attribute.

Examples

```
net = as_sfnetwork(roxel)
sf_attr(net, "agr", active = "edges")
sf_attr(net, "sf_column", active = "nodes")
```

sf_methods

sf methods for sfnetworks

Description

`sf` methods for `sfnetwork` objects.

Usage

```
## S3 method for class 'sfnetwork'
st_as_sf(x, active = NULL, focused = TRUE, ...)

## S3 method for class 'sfnetwork'
st_geometry(obj, active = NULL, focused = TRUE, ...)

## S3 replacement method for class 'sfnetwork'
st_geometry(x) <- value

## S3 method for class 'sfnetwork'
st_drop_geometry(x, ...)

## S3 method for class 'sfnetwork'
st_bbox(obj, active = NULL, ...)

## S3 method for class 'sfnetwork'
st_coordinates(x, active = NULL, ...)
```

```
## S3 method for class 'sfnetwork'
st_is(x, ...)

## S3 method for class 'sfnetwork'
st_is_valid(x, ...)

## S3 method for class 'sfnetwork'
st_as_s2(x, active = NULL, focused = TRUE, ...)

## S3 method for class 'sfnetwork'
st_crs(x, ...)

## S3 replacement method for class 'sfnetwork'
st_crs(x) <- value

## S3 method for class 'sfnetwork'
st_precision(x)

## S3 method for class 'sfnetwork'
st_set_precision(x, precision)

## S3 method for class 'sfnetwork'
st_shift_longitude(x, ...)

## S3 method for class 'sfnetwork'
st_transform(x, ...)

## S3 method for class 'sfnetwork'
st_wrap_dateline(x, ...)

## S3 method for class 'sfnetwork'
st_normalize(x, ...)

## S3 method for class 'sfnetwork'
st_zm(x, ...)

## S3 method for class 'sfnetwork'
st_m_range(obj, active = NULL, ...)

## S3 method for class 'sfnetwork'
st_z_range(obj, active = NULL, ...)

## S3 method for class 'sfnetwork'
st_agr(x, active = NULL, ...)

## S3 replacement method for class 'sfnetwork'
st_agr(x) <- value
```

```
## S3 method for class 'sfnetwork'
st_reverse(x, ...)

## S3 method for class 'sfnetwork'
st_segmentize(x, ...)

## S3 method for class 'sfnetwork'
st_simplify(x, ...)

## S3 method for class 'sfnetwork'
st_join(x, y, ..., ignore_multiple = TRUE)

## S3 method for class 'morphed_sfnetwork'
st_join(x, y, ...)

## S3 method for class 'sfnetwork'
st_filter(x, y, ...)

## S3 method for class 'morphed_sfnetwork'
st_filter(x, y, ...)

## S3 method for class 'sfnetwork'
st_crop(x, y, ...)

## S3 method for class 'morphed_sfnetwork'
st_crop(x, y, ...)

## S3 method for class 'sfnetwork'
st_difference(x, y, ...)

## S3 method for class 'morphed_sfnetwork'
st_difference(x, y, ...)

## S3 method for class 'sfnetwork'
st_intersection(x, y, ...)

## S3 method for class 'morphed_sfnetwork'
st_intersection(x, y, ...)

## S3 method for class 'sfnetwork'
st_intersects(x, y, ...)

## S3 method for class 'sfnetwork'
st_sample(x, ...)

## S3 method for class 'sfnetwork'
st_nearest_points(x, y, ...)
```

```
## S3 method for class 'sfnetwork'
st_area(x, ...)
```

Arguments

x	An object of class sfnetwork .
active	Which network element (i.e. nodes or edges) to activate before extracting. If NULL, it will be set to the current active element of the given network. Defaults to NULL.
focused	Should only features that are in focus be extracted? Defaults to TRUE. See focus for more information on focused networks.
...	Arguments passed on the corresponding sf function.
obj	An object of class sfnetwork .
value	The value to be assigned. See the documentation of the corresponding sf function for details.
precision	The precision to be assigned. See st_precision for details.
y	An object of class sf , or directly convertible to it using st_as_sf . In some cases, it can also be an object of sfg or bbox . Always look at the documentation of the corresponding sf function for details.
ignore_multiple	When performing a spatial join with the nodes table, and there are multiple matches for a single node, only the first one of them is joined into the network. But what should happen with the others? If this argument is set to TRUE, they will be ignored. If this argument is set to FALSE, they will be added as isolated nodes to the returned network. Nodes at equal locations can then be merged using the spatial morpher to_spatial_unique . Defaults to TRUE.

Details

See the [sf](#) documentation. The following methods have a special behavior:

- `st_geometry<-`: The geometry setter requires the replacement geometries to have the same CRS as the network. Node replacements should all be points, while edge replacements should all be linestrings. When replacing node geometries, the boundaries of the edge geometries are replaced as well to preserve the valid spatial network structure. When replacing edge geometries, new edge boundaries that do not match the location of their specified incident node are added as new nodes to the network.
- `st_transform`: No matter if applied to the nodes or edge table, this method will update the coordinates of both tables. The same holds for all other methods that update the way in which the coordinates are encoded without changing their actual location, such as `st_precision`, `st_normalize`, `st_zm`, and others.
- `st_join`: When applied to the nodes table and multiple matches exist for the same node, only the first match is joined. A warning will be given in this case. If `ignore_multiple = FALSE`, multiple matches are instead added as isolated nodes to the returned network.

- `st_intersection`, `st_difference` and `st_crop`: These methods clip edge geometries when applied to the edges table. To preserve a valid spatial network structure, clipped edge boundaries are added as new nodes to the network.
- `st_reverse`: When reversing edge geometries in a directed network, the indices in the from and to columns will be swapped as well.
- `st_segmentize`: When segmentizing edge geometries, the edge boundaries are forced to remain the same such that the valid spatial network structure is preserved. This may lead to slightly inaccurate results.

Geometric unary operations are only supported on `sfnetwork` objects if they do not change the geometry type nor the spatial location of the original features, since that would break the valid spatial network structure. When applying the unsupported operations, first extract the element of interest (nodes or edges) using `st_as_sf`.

Value

The methods for `st_join`, `st_filter`, `st_intersection`, `st_difference` and `st_crop`, as well as the methods for all setter functions and the geometric unary operations preserve the class of the object it is applied to, i.e. either a `sfnetwork` object or its morphed equivalent. When dropping node geometries, an object of class `tbl_graph` is returned. All other methods return the same type of objects as their corresponding sf function. See the `sf` documentation for details.

Examples

```
library(sf, quietly = TRUE)

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1), mfrow = c(1,2))

net = as_sfnetwork(roxel)

# Extract the active network element as sf object.
st_as_sf(net)

# Extract any network element as sf object.
st_as_sf(net, "edges")

# Get the geometry of the active network element.
st_geometry(net)

# Get the geometry of any network element.
st_geometry(net, "edges")

# Replace the geometry of the nodes.
# This will automatically update edge geometries to match the new nodes.
newnet = net
newnds = rep(st_centroid(st_combine(st_geometry(net))), n_nodes(net))
st_geometry(newnet) = newnds

plot(net)
plot(newnet)
```

```

# Drop the geometries of the edges.
# This returns an sfnetwork with spatially implicit edges.
st_drop_geometry(activate(net, "edges"))

# Drop the geometries of the nodes.
# This returns a tbl_graph.
st_drop_geometry(net)

# Get the bounding box of the active network element.
st_bbox(net)

# Get CRS of the network.
st_crs(net)

# Get agr factor of the active network element.
st_agr(net)

# Get agr factor of any network element.
st_agr(net, "edges")

# Spatial join applied to the active network element.
net = st_transform(net, 3035)
codes = st_as_sf(st_make_grid(net, n = c(2, 2)))
codes$post_code = as.character(seq(1000, 1000 + nrow(codes) * 10 - 10, 10))

joined = st_join(net, codes, join = st_intersects)
joined

plot(net, col = "grey")
plot(codes, col = NA, border = "red", lty = 4, lwd = 4, add = TRUE)
text(st_coordinates(st_centroid(st_geometry(codes))), codes$post_code)

plot(st_geometry(joined, "edges"))
plot(st_as_sf(joined, "nodes"), pch = 20, add = TRUE)
par(oldpar)

# Spatial filter applied to the active network element.
p1 = st_point(c(4151358, 3208045))
p2 = st_point(c(4151340, 3207520))
p3 = st_point(c(4151756, 3207506))
p4 = st_point(c(4151774, 3208031))

poly = st_multipoint(c(p1, p2, p3, p4)) |>
  st_cast('POLYGON') |>
  st_sfc(crs = 3035) |>
  st_as_sf()

filtered = st_filter(net, poly, .pred = st_intersects)

plot(net, col = "grey")
plot(poly, border = "red", lty = 4, lwd = 4, add = TRUE)
plot(filtered)

```

```
par(oldpar)
```

sfnetwork

Create a sfnetwork

Description

sfnetwork is a tidy data structure for geospatial networks. It extends the [tbl_graph](#) data structure for relational data into the domain of geospatial networks, with nodes and edges embedded in geographical space, and offers smooth integration with [sf](#) for spatial data analysis.

Usage

```
sfnetwork(
  nodes,
  edges = NULL,
  directed = TRUE,
  node_key = "name",
  edges_as_lines = NULL,
  compute_length = FALSE,
  length_as_weight = deprecated(),
  force = FALSE,
  message = TRUE,
  ...
)
```

Arguments

nodes	The nodes of the network. Should be an object of class sf , or directly convertible to it using st_as_sf . All features should have an associated geometry of type POINT.
edges	The edges of the network. May be an object of class sf , with all features having an associated geometry of type LINESTRING. It may also be a regular data.frame or tbl_df object. In any case, the nodes at the ends of each edge must be referenced in a to and from column, as integers or characters. Integers should refer to the position of a node in the nodes table, while characters should refer to the name of a node stored in the column referred to in the node_key argument. Setting edges to NULL will create a network without edges.
directed	Should the constructed network be directed? Defaults to TRUE.
node_key	The name of the column in the nodes table that character represented to and from columns should be matched against. If NA, the first column is always chosen. This setting has no effect if to and from are given as integers. Defaults to 'name'.

edges_as_lines	Should the edges be spatially explicit, i.e. have LINESTRING geometries stored in a geometry list column? If NULL, this will be automatically defined, by setting the argument to TRUE when the edges are given as an object of class <code>sf</code> , and FALSE otherwise. Defaults to NULL.
compute_length	Should the geographic length of the edges be stored in a column named <code>length</code> ? Uses <code>st_length</code> to compute the length of the edge geometries when edges are spatially explicit, and <code>st_distance</code> to compute the distance between boundary nodes when edges are spatially implicit. If there is already a column named <code>length</code> , it will be overwritten. Please note that the values in this column are not automatically recognized as edge weights. This needs to be specified explicitly when calling a function that uses edge weights. Defaults to FALSE.
length_as_weight	Deprecated, use <code>compute_length</code> instead.
force	Should network validity checks be skipped? Defaults to FALSE, meaning that network validity checks are executed when constructing the network. These checks guarantee a valid spatial network structure. For the nodes, this means that they all should have POINT geometries. In the case of spatially explicit edges, it is also checked that all edges have LINESTRING geometries, nodes and edges have the same CRS and boundary points of edges match their corresponding node coordinates. These checks are important, but also time consuming. If you are already sure your input data meet the requirements, the checks are unnecessary and can be turned off to improve performance.
message	Should informational messages (those messages that are neither warnings nor errors) be printed when constructing the network? Defaults to TRUE.
...	Arguments passed on to <code>st_as_sf</code> , if nodes need to be converted into an <code>sf</code> object during construction.

Value

An object of class `sfnetwork`.

Examples

```
library(sf, quietly = TRUE)

p1 = st_point(c(7, 51))
p2 = st_point(c(7, 52))
p3 = st_point(c(8, 52))
nodes = st_as_sf(st_sfc(p1, p2, p3, crs = 4326))

e1 = st_cast(st_union(p1, p2), "LINESTRING")
e2 = st_cast(st_union(p1, p3), "LINESTRING")
e3 = st_cast(st_union(p3, p2), "LINESTRING")
edges = st_as_sf(st_sfc(e1, e2, e3, crs = 4326))
edges$from = c(1, 1, 3)
edges$to = c(2, 3, 2)

# Default.
sfnetwork(nodes, edges)
```

```
# Undirected network.
sfnetwork(nodes, edges, directed = FALSE)

# Using character encoded from and to columns.
nodes$name = c("city", "village", "farm")
edges$from = c("city", "city", "farm")
edges$to = c("village", "farm", "village")
sfnetwork(nodes, edges, node_key = "name")

# Spatially implicit edges.
sfnetwork(nodes, edges, edges_as_lines = FALSE)

# Store edge lengths in a column named 'length'.
sfnetwork(nodes, edges, compute_length = TRUE)
```

sfnetwork_to_dodgr	<i>Conversion between dodgr streetnets and sfnetworks</i>
--------------------	---

Description

The [dodgr](#) package is designed for routing on directed graphs, and is known for its fast computations of cost matrices, shortest paths, and more. In `sfnetwork`, `dodgr` can be chosen as a routing backend.

Usage

```
dodgr_to_sfnetwork(x, edges_as_lines = TRUE)

sfnetwork_to_dodgr(x, weights = edge_length(), time = FALSE)
```

Arguments

<code>x</code>	For the conversion to <code>sfnetwork</code> : an object of class dodgr_streetnet . For the conversion from <code>sfnetwork</code> : an object of class sfnetwork .
<code>edges_as_lines</code>	Should the created edges be spatially explicit, i.e. have <code>LINESTRING</code> geometries stored in a geometry list column? Defaults to <code>TRUE</code> .
<code>weights</code>	The edge weights to be stored in the <code>dodgr</code> streetnet. Evaluated by evaluate_weight_spec . The default is edge_length , which computes the geographic lengths of the edges. Dual-weights can be provided through dual_weights .
<code>time</code>	Are the provided weights time values? If <code>TRUE</code> , they will be stored in a column named 'time' rather than 'd'. Defaults to <code>FALSE</code> .

Value

For the conversion to `sfnetwork`: An object of class [sfnetwork](#). For the conversion from `sfnetwork`: an object of class [dodgr_streetnet](#).

Note

The [dodgr](#) package is designed for directed graphs. If the provided [sfnetwork](#) object is undirected, it is made directed by duplicating and reversing each edge.

simplify_network	<i>Simplify a spatial network</i>
------------------	-----------------------------------

Description

Construct a simple version of the network. A simple network is defined as a network without loop edges and multiple edges. A loop edge is an edge that starts and ends at the same node. Multiple edges are different edges between the same node pair.

Usage

```
simplify_network(
  x,
  remove_multiple = TRUE,
  remove_loops = TRUE,
  attribute_summary = "first",
  store_original_ids = FALSE,
  store_original_data = FALSE
)
```

Arguments

x	An object of class sfnetwork .
remove_multiple	Should multiple edges be merged into one. Defaults to TRUE.
remove_loops	Should loop edges be removed. Defaults to TRUE.
attribute_summary	How should the attributes of merged multiple edges be summarized? There are several options, see igraph-attribute-combination for details.
store_original_ids	For each group of merged multiple edges, should the indices of the original edges be stored as an attribute of the new edge, in a column named <code>.tidygraph_edge_index</code> ? This is in line with the design principles of tidygraph. Defaults to FALSE.
store_original_data	For each group of merged multiple edges, should the data of the original edges be stored as an attribute of the new edge, in a column named <code>.orig_data</code> ? This is in line with the design principles of tidygraph. Defaults to FALSE.

Value

The simple network as object of class [sfnetwork](#).

Note

When merging groups of multiple edges into a single edge, the geometry of the first edge in each group is preserved. The order of the edges can be influenced by calling [arrange](#) before simplifying.

smooth_pseudo_nodes *Smooth pseudo nodes*

Description

Construct a smoothed version of the network by iteratively removing pseudo nodes, while preserving the connectivity of the network. In the case of directed networks, pseudo nodes are those nodes that have only one incoming and one outgoing edge. In undirected networks, pseudo nodes are those nodes that have two incident edges. Equality of attribute values among the two edges can be defined as an additional requirement by setting the `require_equal` parameter. Connectivity of the network is preserved by concatenating the incident edges of each removed pseudo node.

Usage

```
smooth_pseudo_nodes(
  x,
  protect = NULL,
  require_equal = NULL,
  attribute_summary = "ignore",
  store_original_ids = FALSE,
  store_original_data = FALSE
)
```

Arguments

<code>x</code>	An object of class sfnetwork .
<code>protect</code>	An integer vector of edge indices specifying which nodes should be protected from being removed. Defaults to <code>NULL</code> , meaning that none of the nodes is protected.
<code>require_equal</code>	A character vector of edge column names specifying which attributes of the incident edges of a pseudo node should be equal in order for the pseudo node to be removed? Defaults to <code>NULL</code> , meaning that attribute equality is not considered for pseudo node removal.
<code>attribute_summary</code>	How should the attributes of concatenated edges be summarized? There are several options, see igraph-attribute-combination for details.
<code>store_original_ids</code>	For each concatenated edge, should the indices of the original edges be stored as an attribute of the new edge, in a column named <code>.tidygraph_edge_index</code> ? This is in line with the design principles of <code>tidygraph</code> . Defaults to <code>FALSE</code> .

store_original_data

For each concatenated edge, should the data of the original edges be stored as an attribute of the new edge, in a column named `.orig_data`? This is in line with the design principles of `tidygraph`. Defaults to `FALSE`.

Value

The smoothed network as object of class `sfnetwork`.

spatial_centralities	<i>Compute spatial centralities measures</i>
----------------------	--

Description

These functions are a collection of centrality measures that are specific for spatial networks, and form a spatial extension to [centrality measures](#) in `tidygraph`.

Usage

```
centrality_straightness(...)
```

Arguments

... Additional arguments passed on to other functions.

Details

Just as with all centrality functions in `tidygraph`, these functions are meant to be called inside `tidygraph` verbs such as `mutate` or `filter`, where the network that is currently being worked on is known and thus not needed as an argument to the function. If you want to use an algorithm outside of the `tidygraph` framework you can use `with_graph` to set the context temporarily while the algorithm is being evaluated.

Value

A numeric vector of the same length as the number of nodes in the network.

Functions

- `centrality_straightness()`: The straightness centrality of node *i* is the average ratio of Euclidean distance and network distance between node *i* and all other nodes in the network. ... is forwarded to `st_network_distance` to compute the network distance matrix. Euclidean distances are computed using `st_distance`.

Examples

```
library(tidygraph, quietly = TRUE)

net = as_sfnetwork(roxel, directed = FALSE)

net |>
  activate(nodes) |>
  mutate(sc = centrality_straightness())
```

spatial_edge_measures *Query spatial edge measures*

Description

These functions are a collection of edge measures in spatial networks.

Usage

```
edge_azimuth(degrees = FALSE)

edge_circuitry(Inf_as_NaN = FALSE)

edge_length()

edge_displacement()

edge_segment_count()
```

Arguments

degrees	Should the angle be returned in degrees instead of radians? Defaults to FALSE.
Inf_as_NaN	Should the circuitry values of loop edges be stored as NaN instead of Inf? Defaults to FALSE.

Details

Just as with all query functions in tidygraph, spatial edge measures are meant to be called inside tidygraph verbs such as [mutate](#) or [filter](#), where the network that is currently being worked on is known and thus not needed as an argument to the function. If you want to use an algorithm outside of the tidygraph framework you can use [with_graph](#) to set the context temporarily while the algorithm is being evaluated.

Value

A numeric vector of the same length as the number of edges in the graph.

Functions

- `edge_azimuth()`: The angle in radians between a straight line from the edge startpoint pointing north, and the straight line from the edge startpoint and the edge endpoint. Calculated with `st_geod_azimuth`. Requires a geographic CRS.
- `edge_circuitry()`: The ratio of the length of an edge linestring geometry versus the straight-line distance between its boundary nodes, as described in Giacomini & Levinson, 2015. DOI: 10.1068/b130131p.
- `edge_length()`: The length of an edge linestring geometry as calculated by `st_length`. If edges are spatially implicit, the straight-line distance between its boundary nodes is computed instead, using `st_distance`.
- `edge_displacement()`: The straight-line distance between the two boundary nodes of an edge, as calculated by `st_distance`.
- `edge_segment_count()`: The number of segments contained in the linestring geometry of an edge. Segments are those parts of a linestring geometry that do not contain any interior points.

Examples

```
library(sf, quietly = TRUE)
library(tidygraph, quietly = TRUE)

net = as_sfnetwork(roxel)

net |>
  activate(edges) |>
  mutate(azimuth = edge_azimuth())

net |>
  activate(edges) |>
  mutate(azimuth = edge_azimuth(degrees = TRUE))

net |>
  activate(edges) |>
  mutate(circuitry = edge_circuitry())

net |>
  activate(edges) |>
  mutate(length = edge_length())

net |>
  activate(edges) |>
  mutate(displacement = edge_displacement())

net |>
  activate(edges) |>
  mutate(n_segs = edge_segment_count())
```

`spatial_edge_predicates`*Query edges with spatial predicates*

Description

These functions allow to interpret spatial relations between edges and other geospatial features directly inside `filter` and `mutate` calls. All functions return a logical vector of the same length as the number of edges in the network. Element *i* in that vector is TRUE whenever the chosen spatial predicate applies to the spatial relation between the *i*-th edge and any of the features in *y*.

Usage

```
edge_intersects(y, ...)  
edge_is_disjoint(y, ...)  
edge_touches(y, ...)  
edge_crosses(y, ...)  
edge_is_within(y, ...)  
edge_contains(y, ...)  
edge_contains_properly(y, ...)  
edge_overlaps(y, ...)  
edge_equals(y, ...)  
edge_covers(y, ...)  
edge_is_covered_by(y, ...)  
edge_is_within_distance(y, ...)  
edge_is_nearest(y)
```

Arguments

<code>y</code>	The geospatial features to test the edges against, either as an object of class <code>sf</code> or <code>sfc</code> .
<code>...</code>	Arguments passed on to the corresponding spatial predicate function of <code>sf</code> . See <code>geos_binary_pred</code> . The argument <code>sparse</code> should not be set.

Details

See [geos_binary_pred](#) for details on each spatial predicate. The function `edge_is_nearest` instead wraps around [st_nearest_feature](#) and returns TRUE for element *i* if the *i*-th edge is the nearest edge to any of the features in *y*.

Just as with all query functions in tidygraph, these functions are meant to be called inside tidygraph verbs such as [mutate](#) or [filter](#), where the network that is currently being worked on is known and thus not needed as an argument to the function. If you want to use an algorithm outside of the tidygraph framework you can use [with_graph](#) to set the context temporarily while the algorithm is being evaluated.

Value

A logical vector of the same length as the number of edges in the network.

Note

Note that `edge_is_within_distance` is a wrapper around the `st_is_within_distance` predicate from sf. Hence, it is based on 'as-the-crow-flies' distance, and not on distances over the network.

Examples

```
library(sf, quietly = TRUE)
library(tidygraph, quietly = TRUE)

# Create a network.
net = as_sfnetwork(roxel) |>
  st_transform(3035)

# Create a geometry to test against.
p1 = st_point(c(4151358, 3208045))
p2 = st_point(c(4151340, 3207520))
p3 = st_point(c(4151756, 3207506))
p4 = st_point(c(4151774, 3208031))

poly = st_multipoint(c(p1, p2, p3, p4)) |>
  st_cast('POLYGON') |>
  st_sfc(crs = 3035)

# Use predicate query function in a filter call.
intersects = net |>
  activate(edges) |>
  filter(edge_intersects(poly))

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1))
plot(st_geometry(net, "edges"))
plot(st_geometry(intersects, "edges"), col = "red", lwd = 2, add = TRUE)
par(oldpar)

# Use predicate query function in a mutate call.
net |>
```

```

activate(edges) |>
mutate(disjoint = edge_is_disjoint(poly)) |>
select(disjoint)

# Use predicate query function directly.
intersects = with_graph(net, edge_intersects(poly))
head(intersects)

```

spatial_morphers	<i>Morph spatial networks into a different structure</i>
------------------	--

Description

Spatial morphers form spatial add-ons to the set of [morphers](#) provided by tidygraph. These functions change the existing structure of the network.

Usage

```

to_spatial_contracted(
  x,
  ...,
  simplify = TRUE,
  compute_centroids = TRUE,
  attribute_summary = "ignore",
  summarise_attributes = deprecated(),
  store_original_data = FALSE
)

to_spatial_directed(x)

to_spatial_explicit(x, ...)

to_spatial_implicit(x)

to_spatial_mixed(x, directed)

to_spatial_neighborhood(x, node, threshold, weights = edge_length(), ...)

to_spatial_reversed(x, protect = NULL)

to_spatial_shortest_paths(x, ...)

to_spatial_simple(
  x,
  remove_multiple = TRUE,
  remove_loops = TRUE,
  attribute_summary = "first",

```

```

    summarise_attributes = deprecated(),
    store_original_data = FALSE
)

to_spatial_smooth(
  x,
  protect = NULL,
  require_equal = NULL,
  attribute_summary = "ignore",
  summarise_attributes = deprecated(),
  store_original_data = FALSE
)

to_spatial_subdivision(x, protect = NULL, all = FALSE, merge = TRUE)

to_spatial_subset(x, ..., subset_by = NULL)

to_spatial_transformed(x, ...)

to_spatial_unique(x, attribute_summary = "ignore", store_original_data = FALSE)

```

Arguments

<code>x</code>	An object of class <code>sfnetwork</code> .
<code>...</code>	Arguments to be passed on to other functions. See the description of each morpher for details.
<code>simplify</code>	Should the network be simplified after contraction? Defaults to TRUE. This means that multiple edges and loop edges will be removed. Multiple edges are introduced by contraction when there are several connections between the same groups of nodes. Loop edges are introduced by contraction when there are connections within a group. Note however that setting this to TRUE also removes multiple edges and loop edges that already existed before contraction.
<code>compute_centroids</code>	Should the new geometry of each contracted group of nodes be the centroid of all group members? Defaults to TRUE. If set to FALSE, the geometry of the first node in each group will be used instead, which requires considerably less computing time.
<code>attribute_summary</code>	Whenever groups of nodes or edges are merged into a single feature during morphing, how should their attributes be summarized? There are several options, see igraph-attribute-combination for details.
<code>summarise_attributes</code>	Deprecated, use <code>attribute_summary</code> instead.
<code>store_original_data</code>	Whenever groups of nodes or edges are merged into a single feature during morphing, should the data of the original features be stored as an attribute of the new feature, in a column named <code>.orig_data</code> . This is in line with the design principles of tidygraph. Defaults to FALSE.

directed	Which edges should be directed? Evaluated by evaluate_edge_query .
node	The node for which the neighborhood will be calculated. Evaluated by evaluate_node_query . When multiple nodes are given, only the first one is used.
threshold	The threshold cost to be used. Only nodes reachable within this threshold cost from the reference node will be included in the neighborhood. Should be a numeric value in the same units as the given edge weights. Alternatively, units can be specified explicitly by providing a units object. Multiple threshold values may be given, which will result in multiple neighborhoods being returned.
weights	The edge weights to be used for travel cost computation. Evaluated by evaluate_weight_spec . The default is edge_length , which computes the geographic lengths of the edges.
protect	Nodes or edges to be protected from being changed in structure. Evaluated by evaluate_node_query in the case of nodes and by evaluate_edge_query in the case of edges. Defaults to NULL, meaning that no features are protected.
remove_multiple	Should multiple edges be merged into one. Defaults to TRUE.
remove_loops	Should loop edges be removed. Defaults to TRUE.
require_equal	Which attributes of its incident edges should be equal in order for a pseudo node to be removed? Evaluated as a dplyr_tidy_select argument. Defaults to NULL, meaning that attribute equality is not considered for pseudo node removal.
all	Should edges be subdivided at all their interior points? If set to FALSE, edges are only subdivided at those interior points that share their location with any other interior or boundary point (a node) in the edges table. Defaults to FALSE. By default sfnetworks rounds coordinates to 12 decimal places to determine spatial equality. You can influence this behavior by explicitly setting the precision of the network using st_set_precision .
merge	Should multiple subdivision points at the same location be merged into a single node, and should subdivision points at the same location as an existing node be merged into that node? Defaults to TRUE. If set to FALSE, each subdivision point is added separately as a new node to the network. By default sfnetworks rounds coordinates to 12 decimal places to determine spatial equality. You can influence this behavior by explicitly setting the precision of the network using st_set_precision .
subset_by	Whether to create subgraphs based on nodes or edges.

Details

Morphers are not meant to be called directly. Instead, they should be called inside the [morph](#) verb to change the network structure temporarily. Depending on the chosen morpher, this results in a list of one or more network objects. Single elements of that list can be extracted directly as a new network by calling the morpher inside the [convert](#) verb instead, to make the changes lasting rather than temporary.

It is also possible to create your own morphers. See the documentation of [morph](#) for the requirements for custom morphers.

Value

Either a `morphed_sfnetwork`, which is a list of one or more `sfnetwork` objects, or a `morphed_tbl_graph`, which is a list of one or more `tbl_graph` objects. See the description of each morpher for details.

Functions

- `to_spatial_contracted()`: Combine groups of nodes into a single node per group. ... is forwarded to `group_by` to create the groups. The centroid of such a group will be used by default as geometry of the contracted node. If edges are spatially explicit, edge geometries are updated accordingly such that the valid spatial network structure is preserved. Returns a `morphed_sfnetwork` containing a single element of class `sfnetwork`.
- `to_spatial_directed()`: Make a network directed in the direction given by the linestring geometries of the edges. Differs from `to_directed`, which makes a network directed based on the node indices given in the `from` and `to` columns. In undirected networks these indices may not correspond with the endpoints of the linestring geometries. Returns a `morphed_sfnetwork` containing a single element of class `sfnetwork`. This morpher requires edges to be spatially explicit. If not, use `to_directed`.
- `to_spatial_explicit()`: Create linestring geometries between source and target nodes of edges. If the edges data can be directly converted to an object of class `sf` using `st_as_sf`, extra arguments can be provided as ... and will be forwarded to `st_as_sf` internally. Otherwise, straight lines will be drawn between the source and target node of each edge. Returns a `morphed_sfnetwork` containing a single element of class `sfnetwork`.
- `to_spatial_implicit()`: Drop edge geometries from the network. Returns a `morphed_sfnetwork` containing a single element of class `sfnetwork`.
- `to_spatial_mixed()`: Construct a mixed network in which some edges are directed, and some are undirected. In practice this is implemented as a directed network in which those edges that are meant to be undirected are duplicated and reversed. Returns a `morphed_sfnetwork` containing a single element of class `sfnetwork`.
- `to_spatial_neighborhood()`: Limit a network to the spatial neighborhood of a specific node. ... is forwarded to `st_network_cost` to compute the travel cost from the specified node to all other nodes in the network. Returns a `morphed_sfnetwork` that may contain multiple elements of class `sfnetwork`, depending on the number of given thresholds. When unmorphing only the first instance of both the node and edge data will be used, as the the same node and/or edge can be present in multiple neighborhoods.
- `to_spatial_reversed()`: Reverse the direction of edges. Returns a `morphed_sfnetwork` containing a single element of class `sfnetwork`.
- `to_spatial_shortest_paths()`: Limit a network to those nodes and edges that are part of the shortest path between two nodes. ... is evaluated in the same manner as `st_network_paths`. Returns a `morphed_sfnetwork` that may contain multiple elements of class `sfnetwork`, depending on the number of requested paths. When unmorphing only the first instance of both the node and edge data will be used, as the the same node and/or edge can be present in multiple paths.
- `to_spatial_simple()`: Construct a simple version of the network. A simple network is defined as a network without loop edges and multiple edges. A loop edge is an edge that starts and ends at the same node. Multiple edges are different edges between the same node pair. When merging them into a single edge, the geometry of the first edge is preserved.

The order of the edges can be influenced by calling [arrange](#) before simplifying. Returns a `morphed_sfnetwork` containing a single element of class `sfnetwork`.

- `to_spatial_smooth()`: Construct a smoothed version of the network by iteratively removing pseudo nodes, while preserving the connectivity of the network. In the case of directed networks, pseudo nodes are those nodes that have only one incoming and one outgoing edge. In undirected networks, pseudo nodes are those nodes that have two incident edges. Equality of attribute values among the two edges can be defined as an additional requirement by setting the `require_equal` parameter. Connectivity of the network is preserved by concatenating the incident edges of each removed pseudo node. Returns a `morphed_sfnetwork` containing a single element of class `sfnetwork`.
- `to_spatial_subdivision()`: Construct a subdivision of the network by subdividing edges at interior points. Subdividing means that a new node is added on an edge, and the edge is split in two at that location. Interior points are those points that shape a linestring geometry feature but are not endpoints of it. Returns a `morphed_sfnetwork` containing a single element of class `sfnetwork`. This morpher requires edges to be spatially explicit.
- `to_spatial_subset()`: Subset the network by applying a spatial filter, i.e. a filter on the geometry column based on a spatial predicate. ... is evaluated in the same manner as [st_filter](#). Returns a `morphed_sfnetwork` containing a single element of class `sfnetwork`. For filters on an attribute column, use [to_subgraph](#).
- `to_spatial_transformed()`: Transform the geospatial coordinates of the network into a different coordinate reference system. ... is evaluated in the same manner as [st_transform](#). Returns a `morphed_sfnetwork` containing a single element of class `sfnetwork`.
- `to_spatial_unique()`: Merge nodes with equal geometries into a single node. Returns a `morphed_sfnetwork` containing a single element of class `sfnetwork`. By default `sfnetworks` rounds coordinates to 12 decimal places to determine spatial equality. You can influence this behavior by explicitly setting the precision of the network using [st_set_precision](#).

Examples

```
library(sf, quietly = TRUE)
library(tidygraph, quietly = TRUE)

net = as_sfnetwork(roxel, directed = FALSE) |>
  st_transform(3035)

# Temporary changes with morph and unmorph.
net |>
  activate(edges) |>
  morph(to_spatial_shortest_paths, from = 1, to = 10) |>
  mutate(in_paths = TRUE) |>
  unmorph()

# Lasting changes with convert.
net |>
  activate(edges) |>
  convert(to_spatial_shortest_paths, from = 1, to = 10)
```

spatial_node_predicates*Query nodes with spatial predicates*

Description

These functions allow to interpret spatial relations between nodes and other geospatial features directly inside [filter](#) and [mutate](#) calls. All functions return a logical vector of the same length as the number of nodes in the network. Element *i* in that vector is TRUE whenever the chosen spatial predicate applies to the spatial relation between the *i*-th node and any of the features in *y*.

Usage

```
node_intersects(y, ...)
node_is_disjoint(y, ...)
node_touches(y, ...)
node_is_within(y, ...)
node_equals(y, ...)
node_is_covered_by(y, ...)
node_is_within_distance(y, ...)
node_is_nearest(y)
```

Arguments

<i>y</i>	The geospatial features to test the nodes against, either as an object of class sf or sfc .
<i>...</i>	Arguments passed on to the corresponding spatial predicate function of sf . See geos_binary_pred . The argument <code>sparse</code> should not be set.

Details

See [geos_binary_pred](#) for details on each spatial predicate. The function `node_is_nearest` instead wraps around [st_nearest_feature](#) and returns TRUE for element *i* if the *i*-th node is the nearest node to any of the features in *y*.

Just as with all query functions in `tidygraph`, these functions are meant to be called inside `tidygraph` verbs such as [mutate](#) or [filter](#), where the network that is currently being worked on is known and thus not needed as an argument to the function. If you want to use an algorithm outside of the `tidygraph` framework you can use [with_graph](#) to set the context temporarily while the algorithm is being evaluated.

Value

A logical vector of the same length as the number of nodes in the network.

Note

Note that `node_is_within_distance` is a wrapper around the `st_is_within_distance` predicate from `sf`. Hence, it is based on 'as-the-crow-flies' distance, and not on distances over the network. For distances over the network, use `node_distance_to` with `edge lengths` as `weights` argument.

Examples

```
library(sf, quietly = TRUE)
library(tidygraph, quietly = TRUE)

# Create a network.
net = as_sfnetwork(roxel) |>
  st_transform(3035)

# Create a geometry to test against.
p1 = st_point(c(4151358, 3208045))
p2 = st_point(c(4151340, 3207520))
p3 = st_point(c(4151756, 3207506))
p4 = st_point(c(4151774, 3208031))

poly = st_multipoint(c(p1, p2, p3, p4)) |>
  st_cast('POLYGON') |>
  st_sfc(crs = 3035)

# Use predicate query function in a filter call.
within = net |>
  activate(nodes) |>
  filter(node_is_within(poly))

disjoint = net |>
  activate(nodes) |>
  filter(node_is_disjoint(poly))

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1))
plot(net)
plot(within, col = "red", add = TRUE)
plot(disjoint, col = "blue", add = TRUE)
par(oldpar)

# Use predicate query function in a mutate call.
net |>
  activate(nodes) |>
  mutate(within = node_is_within(poly)) |>
  select(within)

# Use predicate query function directly.
within = with_graph(net, node_is_within(poly))
```

```
head(within)
```

spatial_node_types	<i>Query spatial node types</i>
--------------------	---------------------------------

Description

These functions are a collection of node type queries that are commonly used in spatial network analysis, and form a spatial extension to [node type queries](#) in tidygraph.

Usage

```
node_is_pseudo()
```

```
node_is_dangling()
```

Details

Just as with all query functions in tidygraph, these functions are meant to be called inside tidygraph verbs such as [mutate](#) or [filter](#), where the network that is currently being worked on is known and thus not needed as an argument to the function. If you want to use an algorithm outside of the tidygraph framework you can use [with_graph](#) to set the context temporarily while the algorithm is being evaluated.

Value

A logical vector of the same length as the number of nodes in the network, indicating if each node is of the type in question.

Functions

- `node_is_pseudo()`: Pseudo nodes in directed networks are those nodes with only one incoming and one outgoing edge. In undirected networks pseudo nodes are those nodes with only two incident edges, i.e. nodes of degree 2.
- `node_is_dangling()`: Dangling nodes are nodes with only one incident edge, i.e. nodes of degree 1.

Examples

```
library(sf, quietly = TRUE)
library(tidygraph, quietly = TRUE)

# Create a network.
net = as_sfnetwork(mozart, "mst", directed = FALSE)

# Use query function in a filter call.
pseudos = net |>
```

```

activate(nodes) |>
filter(node_is_pseudo())

danglers = net |>
  activate(nodes) |>
  filter(node_is_dangling())

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1), mfrow = c(1,2))
plot(net, main = "Pseudo nodes")
plot(st_geometry(pseudos), pch = 20, cex = 1.2, col = "orange", add = TRUE)
plot(net, main = "Dangling nodes")
plot(st_geometry(danglers), pch = 20, cex = 1.2, col = "orange", add = TRUE)
par(oldpar)

# Use query function in a mutate call.
net |>
  activate(nodes) |>
  mutate(pseudo = node_is_pseudo(), dangling = node_is_dangling())

# Use query function directly.
danglers = with_graph(net, node_is_dangling())
head(danglers)

```

st_duplicated	<i>Determine duplicated geometries</i>
---------------	--

Description

Determine duplicated geometries

Usage

```
st_duplicated(x)
```

Arguments

x An object of class [sf](#) or [sfc](#).

Value

A logical vector specifying for each feature in x if its geometry is equal to a previous feature in x.

See Also

[duplicated](#)

Examples

```
library(sf, quietly = TRUE)

p1 = st_sfc(st_point(c(1, 1)))
p2 = st_sfc(st_point(c(0, 0)))
p3 = st_sfc(st_point(c(1, 0)))

st_duplicated(c(p1, p2, p2, p3, p1))
```

st_match*Geometry matching*

Description

Geometry matching

Usage

```
st_match(x)
```

Arguments

x An object of class [sf](#) or [sfc](#).

Value

A numeric vector giving for each feature in **x** the position of the first feature in **x** that has an equal geometry.

See Also

[match](#)

Examples

```
library(sf, quietly = TRUE)

p1 = st_sfc(st_point(c(1, 1)))
p2 = st_sfc(st_point(c(0, 0)))
p3 = st_sfc(st_point(c(1, 0)))

st_match(c(p1, p2, p2, p3, p1))
```

st_network_bbox	<i>Compute the bounding box of a spatial network</i>
-----------------	--

Description

A spatial network specific bounding box creator, returning the combined bounding box of the nodes and edges in the network.

Usage

```
st_network_bbox(x, ...)
```

Arguments

x	An object of class sfnetwork .
...	Arguments passed on to st_bbox .

Details

See [st_bbox](#) for details.

Value

The bounding box of the network as an object of class [bbox](#).

Examples

```
library(sf, quietly = TRUE)

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1), mfrow = c(1,2))

# Create a network.
n1 = st_point(c(8, 51))
n2 = st_point(c(7, 51.5))
n3 = st_point(c(8, 52))
n4 = st_point(c(9, 51))
e1 = st_sfc(st_linestring(c(n1, n2, n3)))

nodes = st_as_sf(c(st_sfc(n1), st_sfc(n3), st_sfc(n4)))

edges = st_as_sf(e1)
edges$from = 1
edges$to = 2

net = sfnetwork(nodes, edges)

# Create bounding boxes for nodes, edges and the whole network.
node_bbox = st_bbox(activate(net, "nodes"))
```

```

node_bbox
edge_bbox = st_bbox(activate(net, "edges"))
edge_bbox
net_bbox = st_network_bbox(net)
net_bbox

# Plot.
plot(net, lwd = 2, cex = 4, main = "Element bounding boxes")
plot(st_as_sfc(node_bbox), border = "orange", lty = 2, lwd = 4, add = TRUE)
plot(st_as_sfc(edge_bbox), border = "skyblue", lty = 2, lwd = 4, add = TRUE)

plot(net, lwd = 2, cex = 4, main = "Network bounding box")
plot(st_as_sfc(net_bbox), border = "orange", lty = 2, lwd = 4, add = TRUE)

par(oldpar)

```

st_network_blend	<i>Blend spatial points into a spatial network</i>
------------------	--

Description

Blending a point into a network is the combined process of first projecting the point onto its nearest point on its nearest edge in the network, then subdividing that edge at the location of the projected point, and finally adding the projected point as node to the network. If the location of the projected point is equal an existing node in the network, the attributes of the point will be joined to that node, instead of adding a new node.

Usage

```
st_network_blend(x, y, tolerance = Inf, ignore_duplicates = TRUE)
```

Arguments

x	An object of class sfnetwork .
y	The spatial features to be blended, either as object of class sf or sfc , with POINT geometries.
tolerance	The tolerance distance to be used. Only features that are at least as close to the network as the tolerance distance will be blended. Should be a non-negative number preferably given as an object of class units . Otherwise, it will be assumed that the unit is meters. If set to Inf all features will be blended. Defaults to Inf.
ignore_duplicates	If there are multiple points in y that have the same projected location, only the first one of them is blended into the network. But what should happen with the others? If this argument is set to TRUE, they will be ignored. If this argument is set to FALSE, they will be added as isolated nodes to the returned network. Nodes at equal locations can then be merged using the spatial morpher to_spatial_unique . Defaults to TRUE.

Details

When the projected location of a given point intersects with more than one edge, it is only blended into the first of these edges. Edges are not connected at blending locations. Use the spatial morpher [to_spatial_subdivision](#) for that.

To determine if a projected point is equal to an existing node, and to determine if multiple projected points are equal to each other, sfnetworks by default rounds coordinates to 12 decimal places. You can influence this behavior by explicitly setting the precision of the network using [st_set_precision](#).

Value

The blended network as an object of class [sfnetwork](#).

Note

Due to internal rounding of rational numbers, it may occur that the intersection point between a line and a point is not evaluated as actually intersecting that line by the designated algorithm. Instead, the intersection point lies a tiny-bit away from the edge. Therefore, it is recommended to set the tolerance to a very small number (for example 1e-5) even if you only want to blend points that intersect an edge.

Examples

```
library(sf, quietly = TRUE)

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1), mfrow = c(1,2))

# Create a spatial network.
n1 = st_point(c(0, 0))
n2 = st_point(c(1, 0))
n3 = st_point(c(2, 0))

e1 = st_sfc(st_linestring(c(n1, n2)), crs = 3857)
e2 = st_sfc(st_linestring(c(n2, n3)), crs = 3857)

net = as_sfnetwork(c(e1, e2))

# Create spatial points to blend in.
p1 = st_sfc(st_point(c(0.5, 0.1)))
p2 = st_sfc(st_point(c(0.5, -0.2)))
p3 = st_sfc(st_point(c(1, 0.2)))
p4 = st_sfc(st_point(c(1.75, 0.2)))
p5 = st_sfc(st_point(c(1.25, 0.1)))

pts = st_sf(foo = letters[1:5], geometry = c(p1, p2, p3, p4, p5), crs = 3857)

# Blend all points into the network.
b1 = st_network_blend(net, pts)
b1
```

```

plot(net)
plot(st_geometry(pts), pch = 20, col = "orange", add = TRUE)
plot(b1)
plot(st_geometry(pts), pch = 20, col = "orange", add = TRUE)

# Blend points within a tolerance distance.
tol = units::set_units(0.1, "m")
b2 = st_network_blend(net, pts, tolerance = tol)
b2

plot(net)
plot(st_geometry(pts), pch = 20, col = "orange", add = TRUE)
plot(b2)
plot(st_geometry(pts), pch = 20, col = "orange", add = TRUE)

# Add points with duplicated projected location as isolated nodes.
b3 = st_network_blend(net, pts, ignore_duplicates = FALSE)
b3

par(oldpar)

```

st_network_cost

Compute a cost matrix of a spatial network

Description

Compute total travel costs of shortest paths between nodes in a spatial network.

Usage

```

st_network_cost(
  x,
  from = node_ids(x),
  to = node_ids(x),
  weights = edge_length(),
  direction = "out",
  Inf_as_NaN = FALSE,
  router = getOption("sfndefault_router", "igraph"),
  use_names = FALSE,
  ...
)

st_network_distance(
  x,
  from = node_ids(x),
  to = node_ids(x),
  direction = "out",
  Inf_as_NaN = FALSE,

```

```

router = getOption("sfn_default_router", "igraph"),
use_names = FALSE,
...
)

```

Arguments

x	An object of class sfnetwork .
from	The nodes where the paths should start. Evaluated by evaluate_node_query . By default, all nodes in the network are included.
to	The nodes where the paths should end. Evaluated by evaluate_node_query . By default, all nodes in the network are included.
weights	The edge weights to be used in the shortest path calculation. Evaluated by evaluate_weight_spec . The default is edge_length , which computes the geographic lengths of the edges.
direction	The direction of travel. Defaults to 'out', meaning that the direction given by the network is followed and costs are computed from the points given as argument from. May be set to 'in', meaning that the opposite direction is followed and costs are computed towards the points given as argument from. May also be set to 'all', meaning that the network is considered to be undirected. This argument is ignored for undirected networks.
Inf_as_NaN	Should the cost values of unconnected nodes be stored as NaN instead of Inf? Defaults to FALSE.
router	The routing backend to use for the cost matrix computation. Currently supported options are 'igraph' and 'dodgr'. See Details.
use_names	If a column named name is present in the nodes table, should these names be used as row and column names in the matrix, instead of the node indices? Defaults to FALSE. Ignored when the nodes table does not have a column named name.
...	Additional arguments passed on to the underlying function of the chosen routing backend. See Details.

Details

The sfnetworks package does not implement its own routing algorithms to compute cost matrices. Instead, it relies on "routing backends", i.e. other R packages that have implemented such algorithms. Currently two different routing backends are supported.

The default is [igraph](#). This package supports many-to-many cost matrix computation with the [distances](#) function. The igraph router does not support dual-weighted routing.

The second supported routing backend is [dodgr](#). This package supports many-to-many cost matrix computation with the [dodgr_dists](#) function. It also supports dual-weighted routing. The dodgr package is a conditional dependency of sfnetworks. Using the dodgr router requires the dodgr package to be installed.

The default router can be changed by setting the sfn_default_router option.

Value

An n times m numeric matrix where n is the length of the `from` argument, and m is the length of the `to` argument.

See Also

[st_network_paths](#), [st_network_travel](#)

Examples

```
library(sf, quietly = TRUE)
library(tidygraph, quietly = TRUE)

net = as_sfnetwork(roxel, directed = FALSE) |>
  st_transform(3035)

# Compute the network cost matrix between node pairs.
# Note that geographic edge length is used as edge weights by default.
st_network_cost(net, from = c(495, 121), to = c(495, 121))

# st_network_distance is a synonym for st_network_cost with default weights.
st_network_distance(net, from = c(495, 121), to = c(495, 121))

# Compute the network cost matrix between spatial point features.
# These are snapped to their nearest node before computing costs.
p1 = st_geometry(net, "nodes")[495] + st_sfc(st_point(c(50, -50)))
st_crs(p1) = st_crs(net)
p2 = st_geometry(net, "nodes")[121] + st_sfc(st_point(c(-10, 100)))
st_crs(p2) = st_crs(net)

st_network_cost(net, from = c(p1, p2), to = c(p1, p2))

# Use a node type query function to specify origins and/or destinations.
st_network_cost(net, from = 499, to = node_is_connected(499))

# Use a spatial edge measure to specify edge weights.
# By default edge_length() is used.
st_network_cost(net, c(p1, p2), c(p1, p2), weights = edge_displacement())

# Use a column in the edges table to specify edge weights.
# This uses tidy evaluation.
net |>
  activate("edges") |>
  mutate(foo = runif(n(), min = 0, max = 1)) |>
  st_network_cost(c(p1, p2), c(p1, p2), weights = foo)

# Compute the cost matrix without edge weights.
# Here the cost is defined by the number of edges, ignoring space.
st_network_cost(net, c(p1, p2), c(p1, p2), weights = NA)

# Use the dodgr router for dual-weighted routing.
paths = st_network_cost(net,
```

```

    from = c(p1, p2),
    to = c(p1, p2),
    weights = dual_weights(edge_segment_count(), edge_length()),
    router = "dodgr"
  )

# Not providing any from or to points includes all nodes by default.
with_graph(net, graph_order()) # Our network has 701 nodes.
cost_matrix = st_network_cost(net)
dim(cost_matrix)

```

st_network_faces	<i>Extract the faces of a spatial network</i>
------------------	---

Description

The faces of a spatial network are the areas bounded by edges, without any other edge crossing it. A special face is the outer face, which is the area not bounded by any set of edges.

Usage

```
st_network_faces(x, boundary = NULL)
```

Arguments

x	An object of class sfnetwork .
boundary	The boundary used for the outer face, as an object of class sf or sfc containing a single POLYGON geometry. Note that this boundary should always be larger than the bounding box of the network. If NULL (the default) the network bounding box extended by 0.1 times its diameter is used.

Value

An object of class [sfc](#) with POLYGON geometries, in which each feature represents one face of the network.

Examples

```

library(sf, quietly = TRUE)

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1))

pts = st_transform(mozart, 3035)
net = as_sfnetwork(pts, "delaunay")

faces = st_network_faces(net)

```

```

plot(faces, col = sf.colors(length(faces), categorical = TRUE))
plot(net, add = TRUE)

par(oldpar)

```

st_network_iso

Compute isolines around nodes in a spatial network

Description

Isolines are curves along which a function has a constant value. In spatial networks, they are used to delineate areas that are reachable from a given node within a given travel cost. If the travel cost is distance, they are known as isodistances, while if the travel cost is time, they are known as isochrones. This function finds all network nodes that lie inside an isoline around a specified node.

Usage

```

st_network_iso(
  x,
  node,
  cost,
  weights = edge_length(),
  ...,
  delineate = TRUE,
  ratio = 1,
  allow_holes = FALSE
)

```

Arguments

x	An object of class sfnetwork .
node	The node around which the isolines will be drawn. Evaluated by evaluate_node_query . When multiple nodes are given, only the first one is used.
cost	The constant cost value of the isoline. Should be a numeric value in the same units as the given edge weights. Alternatively, units can be specified explicitly by providing a units object. Multiple values may be given, which will result in multiple isolines being drawn.
weights	The edge weights to be used in the shortest path calculation. Evaluated by evaluate_weight_spec . The default is edge_length , which computes the geographic lengths of the edges.
...	Additional arguments passed on to st_network_cost to compute the cost matrix from the specified node to all other nodes in the network.
delineate	Should the nodes inside the isoline be delineated? If FALSE, the nodes inside the isoline are returned as a MULTIPOINT geometry. If TRUE, the concave hull of that geometry is returned instead. Defaults to TRUE.

ratio	The ratio of the concave hull. Defaults to 1, meaning that the convex hull is computed. See st_concave_hull for details. Ignored if delineate = FALSE. Setting this to a value smaller than 1 requires a GEOS version of at least 3.11.
allow_holes	May the concave hull have holes? Defaults to FALSE. Ignored if delineate = FALSE.

Value

An object of class `sf` with one row per requested isoline. The object contains the following columns:

- `cost`: The constant cost value of the isoline.
- `geometry`: If `delineate = TRUE`, the concave hull of all nodes that lie inside the isoline. Otherwise, those nodes combined into a single MULTIPOINT geometry.

Examples

```
library(sf, quietly = TRUE)

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1))

center = st_centroid(st_combine(st_geometry(roxel)))

net = as_sfnetwork(roxel, directed = FALSE)

iso = net |>
  st_network_iso(node_is_nearest(center), c(1000, 500, 250))

colors = c("#fee6ce90", "#fdae6b90", "#e6550d90")

plot(net)
plot(st_geometry(iso), col = colors, add = TRUE)

# The level of detail can be increased with the ratio argument.
# This requires GEOS >= 3.11.
if (compareVersion(sf_extSoftVersion()[["GEOS"]], "3.11.0") > -1) {

  iso = net |>
    st_network_iso(node_is_nearest(center), c(1000, 500, 250), ratio = 0.3)

  colors = c("#fee6ce90", "#fdae6b90", "#e6550d90")

  plot(net)
  plot(st_geometry(iso), col = colors, add = TRUE)
}

par(oldpar)
```

st_network_join	<i>Join two spatial networks based on equality of node geometries</i>
-----------------	---

Description

A spatial network specific join function which makes a spatial full join on the geometries of the nodes data. Edge data are combined using a [bind_rows](#) semantic, meaning that data are matched by column name and values are filled with NA if missing in either of the networks. The from and to columns in the edge data are updated such that they match the new node indices of the resulting network.

Usage

```
st_network_join(x, y, ...)
```

Arguments

x	An object of class sfnetwork .
y	An object of class sfnetwork , or directly convertible to it using as_sfnetwork .
...	Arguments passed on to graph_join .

Value

The joined networks as an object of class [sfnetwork](#).

Note

By default sfnetworks rounds coordinates to 12 decimal places to determine spatial equality. You can influence this behavior by explicitly setting the precision of the networks using [st_set_precision](#).

Examples

```
library(sf, quietly = TRUE)

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1), mfrow = c(1,2))

# Create two networks.
n1 = st_point(c(0, 0))
n2 = st_point(c(1, 0))
n3 = st_point(c(1,1))
n4 = st_point(c(0,1))

e1 = st_sfc(st_linestring(c(n1, n2)))
e2 = st_sfc(st_linestring(c(n2, n3)))
e3 = st_sfc(st_linestring(c(n3, n4)))

neta = as_sfnetwork(c(e1, e2))
```

```

netb = as_sfnetwork(c(e2, e3))

# Join the networks based on spatial equality of nodes.
net = st_network_join(neta, netb)
net

# Plot.
plot(neta, pch = 15, cex = 2, lwd = 4)
plot(netb, col = "orange", pch = 18, cex = 2, lty = 3, lwd = 4, add = TRUE)
plot(net, cex = 2, lwd = 4)

par(oldpar)

```

st_network_paths	<i>Find shortest paths between nodes in a spatial network</i>
------------------	---

Description

Find shortest paths between nodes in a spatial network

Usage

```

st_network_paths(
  x,
  from,
  to = node_ids(x),
  weights = edge_length(),
  all = FALSE,
  k = 1,
  direction = "out",
  router = getOption("sfndefault_router", "igraph"),
  use_names = FALSE,
  return_cost = TRUE,
  return_geometry = TRUE,
  ...
)

```

Arguments

x	An object of class sfnetwork .
from	The node where the paths should start. Evaluated by evaluate_node_query .
to	The nodes where the paths should end. Evaluated by evaluate_node_query . By default, all nodes in the network are included.
weights	The edge weights to be used in the shortest path calculation. Evaluated by evaluate_weight_spec . The default is edge_length , which computes the geographic lengths of the edges.

all	Should all shortest paths be returned for each pair of nodes? If set to FALSE, only one shortest path is returned for each pair of nodes, even if multiple shortest paths exist. Defaults to FALSE.
k	The number of paths to find. Setting this to any integer higher than 1 returns not only the shortest path, but also the next k - 1 loopless shortest paths, which may be longer than the shortest path. Currently, this is only supported for one-to-one routing, meaning that both the from and to argument should be of length 1. This argument is ignored if all is set to TRUE.
direction	The direction of travel. Defaults to 'out', meaning that the direction given by the network is followed and paths are found from the node given as argument from. May be set to 'in', meaning that the opposite direction is followed and paths are found towards the node given as argument from. May also be set to 'all', meaning that the network is considered to be undirected. This argument is ignored for undirected networks.
router	The routing backend to use for the shortest path computation. Currently supported options are 'igraph' and 'dodgr'. See Details.
use_names	If a column named name is present in the nodes table, should these names be used to encode the nodes in a path, instead of the node indices? Defaults to FALSE. Ignored when the nodes table does not have a column named name.
return_cost	Should the total cost of each path be computed? Defaults to TRUE.
return_geometry	Should a linestring geometry be constructed for each path? Defaults to TRUE. The geometries are constructed by calling st_line_merge on the linestring geometries of the edges in the path. Ignored for networks with spatially implicit edges.
...	Additional arguments passed on to the underlying function of the chosen routing backend. See Details.

Details

The sfnetworks package does not implement its own routing algorithms to find shortest paths. Instead, it relies on "routing backends", i.e. other R packages that have implemented such algorithms. Currently two different routing backends are supported.

The default is [igraph](#). This package supports one-to-many shortest path calculation with the [shortest_paths](#) function. Note that multiple from nodes are not supported. If multiple from nodes are given, only the first one is taken. The igraph router also supports the computation of all shortest path (see the all argument) through the [all_shortest_paths](#) function and of k shortest paths (see the k argument) through the [k_shortest_paths](#) function. In the latter case, only one-to-one routing is supported, meaning that also only one to node should be provided. The igraph router does not support dual-weighted routing.

The second supported routing backend is [dodgr](#). This package supports many-to-many shortest path calculation with the [dodgr_paths](#) function. It also supports dual-weighted routing. The computation of all shortest paths and k shortest paths is currently not supported by the dodgr router. The dodgr package is a conditional dependency of sfnetworks. Using the dodgr router requires the dodgr package to be installed.

The default router can be changed by setting the sfn_default_router option.

Value

An object of class `sf` with one row per requested path. If `return_geometry = FALSE` or edges are spatially implicit, a `tbl_df` is returned instead. If a requested path could not be found, it is included in the output as an empty path.

Depending on the argument settings, the output may include the following columns:

- `from`: The index of the node at the start of the path.
- `to`: The index of the node at the end of the path.
- `node_path`: A vector containing the indices of all nodes on the path, in order of visit.
- `edge_path`: A vector containing the indices of all edges on the path, in order of visit.
- `path_found`: A boolean describing if the requested path exists.
- `cost`: The total cost of the path, obtained by summing the weights of all visited edges. Included if `return_cost = TRUE`.
- `geometry`: The geometry of the path, obtained by merging the geometries of all visited edges. Included if `return_geometry = TRUE` and the network has spatially explicit edges.

See Also

[st_network_cost](#), [st_network_travel](#)

Examples

```
library(sf, quietly = TRUE)
library(tidygraph, quietly = TRUE)

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1))

net = as_sfnetwork(roxel, directed = FALSE) |>
  st_transform(3035)

# Compute the shortest path between two nodes.
# Note that geographic edge length is used as edge weights by default.
paths = st_network_paths(net, from = 495, to = 121)
paths

plot(net, col = "grey")
plot(st_geometry(net)[paths$from], pch = 20, cex = 2, add = TRUE)
plot(st_geometry(paths), col = "orange", lwd = 3, add = TRUE)

# Compute the shortest paths from one to multiple nodes.
# This will return a tibble with one row per path.
paths = st_network_paths(net, from = 495, to = c(121, 131, 141))
paths

plot(net, col = "grey")
plot(st_geometry(net)[paths$from], pch = 20, cex = 2, add = TRUE)
plot(st_geometry(paths), col = "orange", lwd = 3, add = TRUE)
```

```

# Compute the shortest path between two spatial point features.
# These are snapped to their nearest node before finding the path.
p1 = st_geometry(net, "nodes")[495] + st_sfc(st_point(c(50, -50)))
st_crs(p1) = st_crs(net)
p2 = st_geometry(net, "nodes")[121] + st_sfc(st_point(c(-10, 100)))
st_crs(p2) = st_crs(net)

paths = st_network_paths(net, from = p1, to = p2)
paths

plot(net, col = "grey")
plot(c(p1, p2), pch = 20, cex = 2, add = TRUE)
plot(st_geometry(net)[paths$from], pch = 4, cex = 2, add = TRUE)
plot(st_geometry(paths), col = "orange", lwd = 3, add = TRUE)

# Use a node type query function to specify destinations.
st_network_paths(net, 1, node_is_adjacent(1))

# Use a spatial edge measure to specify edge weights.
# By default edge_length() is used.
st_network_paths(net, p1, p2, weights = edge_displacement())

# Use a column in the edges table to specify edge weights.
# This uses tidy evaluation.
net |>
  activate("edges") |>
  mutate(foo = runif(n(), min = 0, max = 1)) |>
  st_network_paths(p1, p2, weights = foo)

# Compute the shortest paths without edge weights.
# This is the path with the fewest number of edges, ignoring space.
st_network_paths(net, p1, p2, weights = NA)

# Use the dodgr router for many-to-many routing.
paths = st_network_paths(net,
  from = c(1, 2),
  to = c(10, 11),
  router = "dodgr"
)

# Use the dodgr router for dual-weighted routing.
paths = st_network_paths(net,
  from = c(1, 2),
  to = c(10, 11),
  weights = dual_weights(edge_segment_count(), edge_length()),
  router = "dodgr"
)

par(oldpar)

```

st_network_travel	<i>Find the optimal route through a set of nodes in a spatial network</i>
-------------------	---

Description

Solve the travelling salesman problem by finding the shortest route through a set of nodes that visits each of those nodes once.

Usage

```
st_network_travel(
  x,
  nodes,
  weights = edge_length(),
  optimizer = "TSP",
  router = getOption("sfn_default_router", "igraph"),
  return_paths = TRUE,
  use_names = FALSE,
  return_cost = TRUE,
  return_geometry = TRUE,
  ...
)
```

Arguments

x	An object of class <code>sfnetwork</code> .
nodes	Nodes to be visited. Evaluated by <code>evaluate_node_query</code> .
weights	The edge weights to be used in the shortest path calculation. Evaluated by <code>evaluate_weight_spec</code> . The default is <code>edge_length</code> , which computes the geographic lengths of the edges.
optimizer	The optimization backend to use for defining the optimal visiting order of the given nodes. Currently the only supported option is 'TSP'. See Details.
router	The routing backend to use for the cost matrix computation and the path computation. Currently supported options are 'igraph' and 'dodgr'. See Details.
return_paths	After defining the optimal visiting order of nodes, should the actual paths connecting those nodes be computed and returned? Defaults to TRUE. If set to FALSE, a vector of indices in visiting order is returned instead, with each index specifying the position of the visited node in the <code>from</code> argument.
use_names	If a column named <code>name</code> is present in the nodes table, should these names be used to encode the nodes in the route, instead of the node indices? Defaults to FALSE. Ignored when the nodes table does not have a column named <code>name</code> and if <code>return_paths = FALSE</code> .
return_cost	Should the total cost of each path between two subsequent nodes be computed? Defaults to TRUE. Ignored if <code>return_paths = FALSE</code> .

`return_geometry` Should a linestring geometry be constructed for each path between two subsequent nodes? Defaults to TRUE. The geometries are constructed by calling [st_line_merge](#) on the linestring geometries of the edges in the path. Ignored if `return_paths = FALSE` and for networks with spatially implicit edges.

`...` Additional arguments passed on to the underlying function of the chosen optimization backend. See Details.

Details

The `sfnetworks` package does not implement its own route optimization algorithms. Instead, it relies on "optimization backends", i.e. other R packages that have implemented such algorithms. Currently the only supported optimization backend to solve the travelling salesman problem is the [TSP](#) package, which provides the [solve_TSP](#) function for this task.

An input for most route optimization algorithms is the matrix containing the travel costs between the nodes to be visited. This is computed using [st_network_cost](#). The output of most route optimization algorithms is the optimal order in which the given nodes should be visited. To compute the actual paths that connect the nodes in that order, the [st_network_paths](#) function is used. Both cost matrix computation and shortest paths computation allow to specify a "routing backend", i.e. an R package that implements algorithms to solve those tasks. See the documentation of the corresponding functions for details.

Value

An object of class `sf` with one row per leg of the optimal route, containing the path of that leg. If `return_geometry = FALSE` or edges are spatially implicit, a `tbl_df` is returned instead. See the documentation of [st_network_paths](#) for details. If `return_paths = FALSE`, a vector of indices in visiting order is returned, with each index specifying the position of the visited node in the `from` argument.

See Also

[st_network_paths](#), [st_network_cost](#)

Examples

```
library(sf, quietly = TRUE)

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1))

net = as_sfnetwork(roxel, directed = FALSE) |>
  st_transform(3035)

# Compute the optimal route through three nodes.
# Note that geographic edge length is used as edge weights by default.
route = st_network_travel(net, c(1, 10, 100))
route

plot(net, col = "grey")
```

```

plot(st_geometry(net)[route$from], pch = 20, cex = 2, add = TRUE)
plot(st_geometry(route), col = "orange", lwd = 3, add = TRUE)

# Instead of returning a path we can return a vector of visiting order.
st_network_travel(net, c(1, 10, 100), return_paths = FALSE)

# Use spatial point features to specify the visiting locations.
# These are snapped to their nearest node before finding the path.
p1 = st_geometry(net, "nodes")[1] + st_sfc(st_point(c(50, -50)))
p2 = st_geometry(net, "nodes")[10] + st_sfc(st_point(c(-10, 100)))
p3 = st_geometry(net, "nodes")[100] + st_sfc(st_point(c(-10, 100)))
pts = c(p1, p2, p3)
st_crs(pts) = st_crs(net)

route = st_network_travel(net, pts)
route

plot(net, col = "grey")
plot(pts, pch = 20, cex = 2, add = TRUE)
plot(st_geometry(net)[route$from], pch = 4, cex = 2, add = TRUE)
plot(st_geometry(route), col = "orange", lwd = 3, add = TRUE)

par(oldpar)

```

st_project_on_network *Project spatial points on a spatial network*

Description

Project spatial points on a spatial network

Usage

```
st_project_on_network(x, network, on = "edges")
```

Arguments

x	The spatial features to be projected, either as object of class sf or sfc , with POINT geometries.
network	An object of class sfnetwork .
on	On what component of the network should the points be projected? Setting it to 'edges' (the default) will find the nearest point on the nearest edge to each point in x. Setting it to 'nodes' will find the nearest node to each point in x.

Details

This function uses [st_nearest_feature](#) to find the nearest edge or node to each feature in x. When projecting on edges, it then finds the nearest point on the nearest edge by calling [st_nearest_points](#) in a pairwise manner.

Value

The same object as `x` but with its geometries replaced by the projections.

Note

Due to internal rounding of rational numbers, even a point projected on an edge may not be evaluated as actually intersecting that edge when calling [st_intersects](#).

Examples

```
library(sf, quietly = TRUE)

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1))

# Create a spatial network.
n1 = st_point(c(0, 0))
n2 = st_point(c(1, 0))
n3 = st_point(c(2, 0))

e1 = st_sfc(st_linestring(c(n1, n2)), crs = 3857)
e2 = st_sfc(st_linestring(c(n2, n3)), crs = 3857)

net = as_sfnetwork(c(e1, e2))

# Create spatial points to project in.
p1 = st_sfc(st_point(c(0.25, 0.1)))
p2 = st_sfc(st_point(c(1, 0.2)))
p3 = st_sfc(st_point(c(1.75, 0.15)))

pts = st_sf(foo = letters[1:3], geometry = c(p1, p2, p3), crs = 3857)

# Project points to the edges of the network.
p1 = st_project_on_network(pts, net)

plot(net)
plot(st_geometry(pts), pch = 20, col = "orange", add = TRUE)
plot(st_geometry(p1), pch = 4, col = "orange", add = TRUE)

# Project points to the nodes of the network.
p2 = st_project_on_network(pts, net, on = "nodes")

plot(net)
plot(st_geometry(pts), pch = 20, col = "orange", add = TRUE)
plot(st_geometry(p2), pch = 4, col = "orange", add = TRUE)

par(oldpar)
```

st_round	<i>Rounding of geometry coordinates</i>
----------	---

Description

Rounding of geometry coordinates

Usage

```
st_round(x, digits = 0)
```

Arguments

x	An object of class sf or sfc .
digits	Integer indicating the number of decimal places to be used.

Value

An object of class [sf](#) or [sfc](#) with rounded coordinates.

See Also

[round](#)

Examples

```
library(sf, quietly = TRUE)

p1 = st_sfc(st_point(c(1.123, 1.123)))
p2 = st_sfc(st_point(c(0.789, 0.789)))
p3 = st_sfc(st_point(c(1.123, 0.789)))

st_round(st_as_sf(c(p1, p2, p2, p3, p1)), digits = 1)
```

subdivide_edges	<i>Subdivide edges at interior points</i>
-----------------	---

Description

Construct a subdivision of the network by subdividing edges at interior points. Subdividing means that a new node is added on an edge, and the edge is split in two at that location. Interior points are those points that shape a linestring geometry feature but are not endpoints of it.

Usage

```
subdivide_edges(x, protect = NULL, all = FALSE, merge = TRUE)
```

Arguments

<code>x</code>	An object of class <code>sfnetwork</code> with spatially explicit edges.
<code>protect</code>	An integer vector of edge indices specifying which edges should be protected from being subdivided. Defaults to <code>NULL</code> , meaning that none of the edges is protected.
<code>all</code>	Should edges be subdivided at all their interior points? If set to <code>FALSE</code> , edges are only subdivided at those interior points that share their location with any other interior or boundary point (a node) in the edges table. Defaults to <code>FALSE</code> .
<code>merge</code>	Should multiple subdivision points at the same location be merged into a single node, and should subdivision points at the same location as an existing node be merged into that node? Defaults to <code>TRUE</code> . If set to <code>FALSE</code> , each subdivision point is added separately as a new node to the network.

Value

The subdivision of `x` as object of class `sfnetwork`.

Note

By default `sfnetworks` rounds coordinates to 12 decimal places to determine spatial equality. You can influence this behavior by explicitly setting the precision of the network using `st_set_precision`.

<code>tidygraph_methods</code>	<i>tidygraph methods for sfnetworks</i>
--------------------------------	---

Description

Normally tidygraph functions should work out of the box on `sfnetwork` objects, but in some cases special treatment is needed especially for the geometry column, requiring a specific method.

Usage

```
## S3 method for class 'sfnetwork'
as_tbl_graph(x, ...)

## S3 method for class 'sfnetwork'
reroute(.data, ...)

## S3 method for class 'sfnetwork'
morph(.data, ...)

## S3 method for class 'morphed_sfnetwork'
unmorph(.data, ...)
```

Arguments

x	An object of class <code>sfnetwork</code> .
...	Arguments passed on the corresponding tidygraph function.
.data	An object of class <code>sfnetwork</code> .

Details

See the [tidygraph](#) documentation. The following methods have a special behavior:

- `reroute`: To preserve the valid spatial network structure, this method will replace the boundaries of edge geometries by the location of the node those edges are rerouted to or from. Note that when the goal is to reverse edges in a spatial network, `reroute` will not simply reverse the edge geometries. In that case it is recommended to use the `sfnetwork` method for `st_reverse` instead.
- `morph`: This method checks if the morphed network still has spatially embedded nodes. In that case a `morphed_sfnetwork` is returned. If not, a `morphed_tbl_graph` is returned instead.
- `unmorph`: This method makes sure the geometry list column is correctly handled during the unmorphing process.

Value

The method for `as_tbl_graph` returns an object of class `tbl_graph`. The method for `morph` returns a `morphed_sfnetwork` if the morphed network is still spatial, and a `morphed_tbl_graph` otherwise. All other methods return an object of class `sfnetwork`.

validate_network	<i>Validate the structure of a sfnetwork</i>
------------------	--

Description

Validate the structure of a `sfnetwork`

Usage

```
validate_network(x, message = TRUE)
```

Arguments

x	An object of class <code>sfnetwork</code> .
message	Should messages be printed during validation? Defaults to TRUE.

Details

A valid `sfnetwork` structure means that all nodes have POINT geometries, and - when edges are spatially explicit - all edges have LINESTRING geometries, nodes and edges have the same coordinate reference system and the same coordinate precision, and coordinates of edge boundaries match coordinates of their corresponding nodes.

Value

Nothing when the network is valid. Otherwise, an error is thrown.

wrap_igraph	<i>Run an igraph function on an sfnetwork object</i>
-------------	--

Description

Since `sfnetwork` objects inherit `igraph` objects, any `igraph` function can be called on a `sfnetwork`. However, if this function returns a network, it will be an `igraph` object rather than a `sfnetwork` object. With `wrap_igraph`, such a function will preserve the `sfnetwork` class, after checking if the network returned by `igraph` still has a valid spatial network structure.

Usage

```
wrap_igraph(.data, .f, ..., .force = FALSE, .message = TRUE)
```

Arguments

<code>.data</code>	An object of class <code>sfnetwork</code> .
<code>.f</code>	An function from the <code>igraph</code> package that accepts a graph as its first argument, and returns a graph.
<code>...</code>	Arguments passed on to <code>.f</code> .
<code>.force</code>	Should network validity checks be skipped? Defaults to <code>FALSE</code> , meaning that network validity checks are executed when returning the new network. These checks guarantee a valid spatial network structure. For the nodes, this means that they all should have <code>POINT</code> geometries. In the case of spatially explicit edges, it is also checked that all edges have <code>LINESTRING</code> geometries, nodes and edges have the same CRS and boundary points of edges match their corresponding node coordinates. These checks are important, but also time consuming. If you are already sure your input data meet the requirements, the checks are unnecessary and can be turned off to improve performance.
<code>.message</code>	Should informational messages (those messages that are neither warnings nor errors) be printed when constructing the network? Defaults to <code>TRUE</code> .

Value

An object of class `sfnetwork`.

Examples

```
oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1), mfrow = c(1,2))

net = as_sfnetwork(mozart, "delanay", directed = FALSE)
mst = wrap_igraph(net, igraph::mst, .message = FALSE)
```

```
mst
```

```
plot(net)
```

```
plot(mst)
```

```
par(oldpar)
```

Index

- * **datasets**
 - mozart, [26](#)
 - roxel, [35](#)
- all_shortest_paths, [73](#)
- arrange, [46](#), [56](#)
- as.integer, [13](#), [17](#), [18](#)
- as.linnet, [3](#)
- as.logical, [13](#)
- as_s2_geography, [4](#)
- as_sfnetwork, [3](#), [4](#), [71](#)
- as_tbl_graph, [5](#), [82](#)
- as_tbl_graph.sfnetwork
(tidygraph_methods), [81](#)
- as_tibble, [7](#), [7](#)
- autoplot, [8](#), [8](#)
- bbox, [32](#), [39](#), [62](#)
- bind_edges, [9](#)
- bind_nodes, [9](#)
- bind_rows, [9](#), [71](#)
- bind_spatial, [9](#)
- bind_spatial_edges (bind_spatial), [9](#)
- bind_spatial_nodes (bind_spatial), [9](#)
- centrality_straightness
(spatial_centrality), [47](#)
- contract_nodes, [10](#)
- convert, [54](#)
- create_from_spatial_lines, [5](#), [6](#), [11](#), [14](#)
- create_from_spatial_points, [5](#), [12](#), [12](#)
- data, [15](#)
- data.frame, [42](#)
- dbscan, [20](#)
- distances, [66](#)
- dodgr, [5](#), [16](#), [44](#), [45](#), [66](#), [73](#)
- dodgr_dists, [66](#)
- dodgr_paths, [73](#)
- dodgr_streetnet, [5](#), [44](#)
- dodgr_to_sfnetwork, [5](#)
- dodgr_to_sfnetwork
(sfnetwork_to_dodgr), [44](#)
- dplyr_tidy_select, [54](#)
- dual_weights, [16](#), [19](#), [44](#)
- duplicated, [60](#)
- edge predicate query function, [17](#)
- edge type query function, [17](#)
- edge_azimuth (spatial_edge_measures), [48](#)
- edge_circuitry (spatial_edge_measures),
[48](#)
- edge_contains
(spatial_edge_predicates), [50](#)
- edge_contains_properly
(spatial_edge_predicates), [50](#)
- edge_covers (spatial_edge_predicates),
[50](#)
- edge_crosses (spatial_edge_predicates),
[50](#)
- edge_data (data), [15](#)
- edge_displacement
(spatial_edge_measures), [48](#)
- edge_equals (spatial_edge_predicates),
[50](#)
- edge_ids (ids), [20](#)
- edge_intersects
(spatial_edge_predicates), [50](#)
- edge_is_covered_by
(spatial_edge_predicates), [50](#)
- edge_is_disjoint
(spatial_edge_predicates), [50](#)
- edge_is_nearest
(spatial_edge_predicates), [50](#)
- edge_is_within
(spatial_edge_predicates), [50](#)
- edge_is_within_distance
(spatial_edge_predicates), [50](#)
- edge_length, [44](#), [54](#), [66](#), [69](#), [72](#), [76](#)
- edge_length (spatial_edge_measures), [48](#)

- edge_overlaps
 - (spatial_edge_predicates), 50
- edge_segment_count
 - (spatial_edge_measures), 48
- edge_touches (spatial_edge_predicates), 50
- evaluate_edge_query, 16, 54
- evaluate_node_query, 17, 54, 66, 69, 72, 76
- evaluate_weight_spec, 16, 18, 44, 54, 66, 69, 72, 76
- filter, 20, 31, 47, 48, 50, 51, 57, 59
- focus, 4, 7, 15, 21, 27, 29, 30, 39
- geos_binary_pred, 50, 51, 57
- ggplot, 8
- graph_join, 71
- group_by, 55
- group_spatial, 19
- group_spatial_dbSCAN (group_spatial), 19
- grouping, 19
- ids, 20
- igraph, 27, 66, 73, 83
- is_sfnetwork (is_sfnetwork), 21
- is_sfnetwork, 21
- k_shortest_paths, 73
- linnet, 3, 5
- make_edges_directed, 22
- make_edges_explicit, 23
- make_edges_follow_indices, 23, 26
- make_edges_implicit, 24
- make_edges_mixed, 25
- make_edges_valid, 25
- match, 61
- morph, 54, 82
- morph.sfnetwork (tidygraph_methods), 81
- morphers, 52
- mozart, 26
- mutate, 20, 31, 47, 48, 50, 51, 57, 59
- n, 27
- n_edges (n), 27
- n_nodes (n), 27
- nb, 27
- nb_to_sfnetwork (nb), 27
- nearest, 29
- nearest_edge_ids (nearest_ids), 30
- nearest_edges (nearest), 29
- nearest_ids, 30
- nearest_node_ids (nearest_ids), 30
- nearest_nodes (nearest), 29
- node predicate query function, 18
- node type query function, 18
- node_coordinates, 31
- node_data (data), 15
- node_distance_to, 58
- node_equals (spatial_node_predicates), 57
- node_ids (ids), 20
- node_intersects
 - (spatial_node_predicates), 57
- node_is_covered_by
 - (spatial_node_predicates), 57
- node_is_dangling (spatial_node_types), 59
- node_is_disjoint
 - (spatial_node_predicates), 57
- node_is_nearest
 - (spatial_node_predicates), 57
- node_is_pseudo (spatial_node_types), 59
- node_is_within
 - (spatial_node_predicates), 57
- node_is_within_distance
 - (spatial_node_predicates), 57
- node_M (node_coordinates), 31
- node_touches (spatial_node_predicates), 57
- node_X (node_coordinates), 31
- node_Y (node_coordinates), 31
- node_Z (node_coordinates), 31
- play_geometric, 14
- play_geometric (play_spatial), 32
- play_spatial, 32
- plot.sf, 34
- plot.sfnetwork, 33
- psp, 6
- quosure, 17–19
- reroute.sfnetwork (tidygraph_methods), 81
- round, 80
- roxel, 35
- s2_geography, 4

- `sf`, 5–7, 9, 11, 13, 15, 17, 18, 26, 28–30, 32, 35, 36, 39, 40, 42, 43, 50, 55, 57, 60, 61, 63, 68, 70, 74, 77, 78, 80
- `sf_attr`, 35
- `sf_methods`, 36
- `sfc`, 5, 11, 13, 17, 18, 28–30, 32, 50, 57, 60, 61, 63, 68, 78, 80
- `sfg`, 32, 39
- `sfnetwork`, 3–12, 14, 15, 17–19, 21–30, 33, 36, 39, 40, 42, 44–47, 53, 55, 56, 62–64, 66, 68, 69, 71, 72, 76, 78, 81–83
- `sfnetwork_to_dodgr`, 44
- `sfnetwork_to_nb` (nb), 27
- `sGBP`, 13, 27
- `shortest_paths`, 73
- `simplify_network`, 45
- `smooth_pseudo_nodes`, 46
- `solve_TSP`, 77
- `spatial edge measure function`, 19
- `spatial_centrality`, 47
- `spatial_edge_measures`, 48
- `spatial_edge_predicates`, 50
- `spatial_morphers`, 52
- `spatial_node_predicates`, 57
- `spatial_node_types`, 59
- `spatstat.geom`, 5
- `st_agr.sfnetwork` (sf_methods), 36
- `st_agr<-`.sfnetwork (sf_methods), 36
- `st_area.sfnetwork` (sf_methods), 36
- `st_as_s2.sfnetwork` (sf_methods), 36
- `st_as_sf`, 5, 6, 23, 39, 40, 42, 43, 55
- `st_as_sf.sfnetwork` (sf_methods), 36
- `st_bbox`, 62
- `st_bbox.sfnetwork` (sf_methods), 36
- `st_concave_hull`, 70
- `st_coordinates.sfnetwork` (sf_methods), 36
- `st_crop`, 40
- `st_crop.morphed_sfnetwork` (sf_methods), 36
- `st_crop.sfnetwork` (sf_methods), 36
- `st_crs.sfnetwork` (sf_methods), 36
- `st_crs<-`.sfnetwork (sf_methods), 36
- `st_difference`, 40
- `st_difference.morphed_sfnetwork` (sf_methods), 36
- `st_difference.sfnetwork` (sf_methods), 36
- `st_distance`, 13, 20, 32, 43, 47, 49
- `st_drop_geometry.sfnetwork` (sf_methods), 36
- `st_duplicated`, 60
- `st_filter`, 40, 56
- `st_filter.morphed_sfnetwork` (sf_methods), 36
- `st_filter.sfnetwork` (sf_methods), 36
- `st_geod_azimuth`, 49
- `st_geometry.sfnetwork` (sf_methods), 36
- `st_geometry<-`.sfnetwork (sf_methods), 36
- `st_intersection`, 40
- `st_intersection.morphed_sfnetwork` (sf_methods), 36
- `st_intersection.sfnetwork` (sf_methods), 36
- `st_intersects`, 79
- `st_intersects.sfnetwork` (sf_methods), 36
- `st_is.sfnetwork` (sf_methods), 36
- `st_is_valid.sfnetwork` (sf_methods), 36
- `st_join`, 40
- `st_join.morphed_sfnetwork` (sf_methods), 36
- `st_join.sfnetwork` (sf_methods), 36
- `st_length`, 11, 13, 32, 43, 49
- `st_line_merge`, 73, 77
- `st_m_range.sfnetwork` (sf_methods), 36
- `st_match`, 61
- `st_nearest_feature`, 17, 18, 29, 30, 51, 57, 78
- `st_nearest_points`, 78
- `st_nearest_points.sfnetwork` (sf_methods), 36
- `st_network_bbox`, 62
- `st_network_blend`, 63
- `st_network_cost`, 55, 65, 69, 74, 77
- `st_network_distance`, 20, 47
- `st_network_distance` (st_network_cost), 65
- `st_network_faces`, 68
- `st_network_iso`, 69
- `st_network_join`, 71
- `st_network_paths`, 55, 67, 72, 77
- `st_network_travel`, 67, 74, 76
- `st_normalize.sfnetwork` (sf_methods), 36
- `st_precision`, 39
- `st_precision.sfnetwork` (sf_methods), 36
- `st_project_on_network`, 78

- st_reverse, [82](#)
- st_reverse.sfnetwork(sf_methods), [36](#)
- st_round, [80](#)
- st_sample, [32](#)
- st_sample.sfnetwork(sf_methods), [36](#)
- st_segmentize.sfnetwork(sf_methods), [36](#)
- st_set_precision, [12](#), [54](#), [56](#), [64](#), [71](#), [81](#)
- st_set_precision.sfnetwork(sf_methods), [36](#)
- st_shift_longitude.sfnetwork(sf_methods), [36](#)
- st_simplify.sfnetwork(sf_methods), [36](#)
- st_transform, [56](#)
- st_transform.sfnetwork(sf_methods), [36](#)
- st_wrap_dateline.sfnetwork(sf_methods), [36](#)
- st_z_range.sfnetwork(sf_methods), [36](#)
- st_zm.sfnetwork(sf_methods), [36](#)
- subdivide_edges, [80](#)
- tbl_df, [42](#), [74](#), [77](#)
- tbl_graph, [5](#), [6](#), [40](#), [42](#), [55](#), [82](#)
- tibble, [7](#), [15](#)
- tidygraph, [82](#)
- tidygraph_methods, [81](#)
- to_directed, [55](#)
- to_spatial_contracted(spatial_morphers), [52](#)
- to_spatial_directed(spatial_morphers), [52](#)
- to_spatial_explicit(spatial_morphers), [52](#)
- to_spatial_implicit(spatial_morphers), [52](#)
- to_spatial_mixed(spatial_morphers), [52](#)
- to_spatial_neighborhood(spatial_morphers), [52](#)
- to_spatial_reversed(spatial_morphers), [52](#)
- to_spatial_shortest_paths(spatial_morphers), [52](#)
- to_spatial_simple(spatial_morphers), [52](#)
- to_spatial_smooth(spatial_morphers), [52](#)
- to_spatial_subdivision, [64](#)
- to_spatial_subdivision(spatial_morphers), [52](#)
- to_spatial_subset(spatial_morphers), [52](#)
- to_spatial_transformed(spatial_morphers), [52](#)
- to_spatial_unique, [39](#), [63](#)
- to_spatial_unique(spatial_morphers), [52](#)
- to_subgraph, [56](#)
- TSP, [77](#)
- units, [33](#), [54](#), [63](#), [69](#)
- unmorph.morphed_sfnetwork(tidygraph_methods), [81](#)
- validate_network, [82](#)
- with_graph, [20](#), [31](#), [47](#), [48](#), [51](#), [57](#), [59](#)
- wrap_igraph, [83](#), [83](#)