Package ‘sfnetworks’

December 6, 2024

Title Tidy Geospatial Networks

Version 0.6.5

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 crayon, dplyr, graphics, igraph, lwgeom, rlang, sf, sfheaders, tibble, tidygraph, units, utils

Suggests dbscan, fansi, ggplot2 (>= 3.0.0), knitr, purrr, rmarkdown, s2 (>= 1.0.1), spatstat.geom, spatstat.linnet, testthat, TSP

VignetteBuilder knitr

ByteCompile true

Encoding UTF-8

LazyData true

RoxygenNote 7.3.2

NeedsCompilation no

Author Lucas van der Meer [aut, cre] (<https://orcid.org/0000-0001-6336-8628>), Lorena Abad [aut] (<https://orcid.org/0000-0003-0554-734X>), Andrea Gilardi [aut] (<https://orcid.org/0000-0002-9424-7439>), Robin Lovelace [aut] (<https://orcid.org/0000-0001-5679-6536>)

Repository CRAN

Date/Publication 2024-12-06 15:40:02 UTC

Contents

as.linnet . . . 2
as_sfnetwork . . . 3
as_tibble . . . 5
autoplot . . . 6
is.sfnetwork . . . 6
node_coordinates . . . 7
plot.sfnetwork . . . 8
roxel . . . 9
s2 . . . 10
sf . . . 10
sfnetwork . . . 14
sf_attr . . . 17
spatial_edge_measures . . . 17
spatial_edge_predicates . . . 19
spatial_morphers . . . 21
spatial_node_predicates . . . 25
st_network_bbox . . . 27
st_network_blend . . . 28
st_network_cost . . . 30
st_network_join . . . 33
st_network_paths . . . 34

Index **38**

`as.linnet` *Convert a sfnetwork into a linnet*

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_sfnetwork *Convert a foreign object to a sfnetwork*

Description

Convert a given object into an object of class sfnetwork. If an object can be read by as_tbl_graph and the nodes can be read by st_as_sf, it is automatically supported.

Usage

```
as_sfnetwork(x, ...)

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

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

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

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

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

## S3 method for class 'sfNetwork'
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 an sfnetwork.
...	Arguments passed on to the sfnetwork construction function.

Value

An object of class sfnetwork.

Methods (by class)

- `as_sfnetwork(sf)`: Only sf objects with either exclusively geometries of type `LINESTRING` or exclusively geometries of type `POINT` are supported. For lines, is assumed that the given features form the edges. Nodes are created at the endpoints of the lines. Endpoints which are shared between multiple edges become a single node. For points, it is assumed that the given features geometries form the nodes. They will be connected by edges sequentially. Hence, point 1 to point 2, point 2 to point 3, etc.

Examples

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

# With LINESTRING geometries.
as_sfnetwork(roxel)

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1), mfrow = c(1,2))
plot(st_geometry(roxel))
plot(as_sfnetwork(roxel))
par(oldpar)

# With POINT geometries.
p1 = st_point(c(7, 51))
p2 = st_point(c(7, 52))
p3 = st_point(c(8, 52))
points = st_as_sf(st_sfc(p1, p2, p3))
as_sfnetwork(points)

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1), mfrow = c(1,2))
plot(st_geometry(points))
plot(as_sfnetwork(points))
par(oldpar)

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

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

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, spatial = TRUE, ...)
```

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`.
`spatial`	Should the extracted tibble be a 'spatial tibble', i.e. an object of class `c('sf', 'tbl_df')`, if it contains a geometry list column. Defaults to `TRUE`.
`...`	Arguments passed on to `as_tibble`.

Value

The active element of the network as an object of class `tibble`.

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

`object`	An object of class `sfnetwork`.
`...`	Ignored.

Details

See `autoplot`.

Value

An object of class `ggplot`.

is.sfnetwork — *Check if an object is a sfnetwork*

Description

Check if an object is a sfnetwork

Usage

```
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))
```

---

node_coordinates *Query node coordinates*

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())
```

`plot.sfnetwork` *Plot sfnetwork geometries*

Description

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

Usage

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

Arguments

`x`	Object of class `sfnetwork`.
`draw_lines`	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.
`...`	Arguments passed on to `plot.sf`

Details

This is a basic plotting functionality. For more advanced plotting, it is recommended to extract the nodes and edges from the network, and plot them separately with one of the many available spatial plotting functions as can be found in `sf`, `tmap`, `ggplot2`, `ggspatial`, and others.

Value

This is a plot method and therefore has no visible return value.

Examples

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

# When lines are spatially implicit.
par(mar = c(1,1,1,1), mfrow = c(1,2))
net = as_sfnetwork(roxel, edges_as_lines = FALSE)
plot(net)
plot(net, draw_lines = FALSE)

# Changing default settings.
par(mar = c(1,1,1,1), mfrow = c(1,1))
plot(net, col = 'blue', pch = 18, lwd = 1, cex = 2)

# Add grid and axis
par(mar = c(2.5,2.5,1,1))
plot(net, graticule = TRUE, axes = 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'. The topology is cleaned with the v.clean tool in GRASS GIS.

Usage

```
roxel
```

Format

An object of class `sf` with `LINESTRING` geometries, containing 851 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

`s2` *s2 methods for sfnetworks*

Description

s2 methods for sfnetworks

Usage

```
as_s2_geography.sfnetwork(x, ...)
```

Arguments

`x`	An object of class `sfnetwork`.
`...`	Arguments passed on the corresponding `s2` function.

`sf` *sf methods for sfnetworks*

Description

`sf` methods for `sfnetwork` objects.

Usage

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

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

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

## 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_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_simplify(x, ...)

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

## 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.
...	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.

Details

See the sf documentation.

Value

The sfnetwork method for st_as_sf returns the active element of the network as object of class sf. The sfnetwork and morphed_sfnetwork methods for st_join, st_filter, st_intersection, st_difference, st_crop and the setter functions return an object of class sfnetwork and morphed_sfnetwork respectively. 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)

net = as_sfnetwork(roxel)

# Extract the active network element.
st_as_sf(net)

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

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

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

# Get bbox 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

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

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1), mfrow = c(1,2))
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,
  length_as_weight = FALSE,
  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 either be encoded 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 encoded 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 sf, and FALSE otherwise. Defaults to NULL.
length_as_weight	Should the length of the edges be stored in a column named weight? If set to TRUE, this will calculate the length of the linestring geometry of the edge in the case of spatially explicit edges, and the straight-line distance between the source and target node in the case of spatially implicit edges. If there is already a column named weight, it will be overwritten. Defaults to FALSE.
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 st_as_sf, if nodes need to be converted into an sf object during construction.

Value

An object of class sfnetwork.

Examples

```
library(sf, quietly = TRUE)

## Create sfnetwork from sf objects
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 lenghts in a weight column.
sfnetwork(nodes, edges, length_as_weight = TRUE)

# Adjust the number of features printed by active and inactive components
oldoptions = options(sfn_max_print_active = 1, sfn_max_print_inactive = 2)
sfnetwork(nodes, edges)
options(oldoptions)
```

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

`x`	An object of class `sfnetwork`.
`name`	Name of the attribute to query. Either `'sf_column'` or `'agr'`.
`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`.

Details

sf attributes include `sf_column` (the name of the sf column) and `agr` (the attribute-geometry-relationships).

Value

The value of the attribute matched, or `NULL` if no exact match is found.

Examples

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

spatial_edge_measures — *Query spatial edge measures*

Description

These functions are a collection of specific spatial edge measures, that form a spatial extension to edge measures in `tidygraph`.

Usage

```
edge_azimuth(degrees = FALSE)

edge_circuity(Inf_as_NaN = FALSE)

edge_length()

edge_displacement()
```

Arguments

degrees	Should the angle be returned in degrees instead of radians? Defaults to `FALSE`.
Inf_as_NaN	Should the circuity 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_circuity()`: The ratio of the length of an edge linestring geometry versus the straight-line distance between its boundary nodes, as described in Giacomin & Levinson, 2015. DOI: 10.1068/b130131p.
- `edge_length()`: The length of an edge linestring geometry as calculated by `st_length`.
- `edge_displacement()`: The straight-line distance between the two boundary nodes of an edge, as calculated by `st_distance`.

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(circuity = edge_circuity())

net %>%
  activate("edges") %>%
  mutate(length = edge_length())

net %>%
  activate("edges") %>%
  mutate(displacement = edge_displacement())
```

---

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 `any(predicate(x[i], y[j]))` is `TRUE`. Hence, in the case of using `edge_intersects`, element i in the returned vector is `TRUE` when edge i intersects with any of the features given 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, ...)
```

Arguments

y — The geospatial features to test the edges against, either as an object of class sf or sfc.

... — Arguments passed on to the corresponding spatial predicate function of sf. See geos_binary_pred.

Details

See geos_binary_pred for details on each spatial predicate. 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)
```

`spatial_morphers` *Spatial morphers for sfnetworks*

Description

Spatial morphers form spatial add-ons to the set of `morphers` provided by `tidygraph`. These functions are not meant to be called directly. They should either be passed into `morph` to create a temporary alternative representation of the input network. Such an alternative representation is a list of one or more network objects. Single elements of that list can be extracted directly as a new network by passing the morpher to `convert` instead, to make the changes lasting rather than temporary. Alternatively, if the morphed state contains multiple elements, all of them can be extracted together inside a `tbl_df` by passing the morpher to `crystallise`.

Usage

```
to_spatial_contracted(
  x,
  ...,
  simplify = FALSE,
  summarise_attributes = "ignore",
  store_original_data = FALSE
)

to_spatial_directed(x)

to_spatial_explicit(x, ...)

to_spatial_neighborhood(x, node, threshold, weights = NULL, from = TRUE, ...)
```

```
to_spatial_shortest_paths(x, ...)

to_spatial_simple(
  x,
  remove_multiple = TRUE,
  remove_loops = TRUE,
  summarise_attributes = "first",
  store_original_data = FALSE
)

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

to_spatial_subdivision(x)

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

to_spatial_transformed(x, ...)
```

Arguments

`x` An object of class `sfnetwork`.

`...` Arguments to be passed on to other functions. See the description of each morpher for details.

`simplify` Should the network be simplified after contraction? 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. Defaults to `FALSE`.

`summarise_attributes` Whenever multiple features (i.e. nodes and/or edges) are merged into a single feature during morphing, how should their attributes be combined? Several options are possible, see `igraph-attribute-combination` for details.

`store_original_data` Whenever multiple features (i.e. nodes and/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 `.orig_data`. This is in line with the design principles of `tidygraph`. Defaults to `FALSE`.

`node` The geospatial point for which the neighborhood will be calculated. Can be an integer, referring to the index of the node for which the neighborhood will be calculated. Can also be an object of class `sf` or `sfc`, containing a single feature.

In that case, this point will be snapped to its nearest node before calculating the neighborhood. When multiple indices or features are given, only the first one is taken.

threshold
: The threshold distance to be used. Only nodes within the threshold distance from the reference node will be included in the neighborhood. Should be a numeric value in the same units as the weight values used for distance calculation.

weights
: The edge weights used to calculate distances on the network. Can be a numeric vector giving edge weights, or a column name referring to an attribute column in the edges table containing those weights. If set to NULL, the values of a column named weight in the edges table will be used automatically, as long as this column is present. If not, the geographic edge lengths will be calculated internally and used as weights.

from
: Should distances be calculated from the reference node towards the other nodes? Defaults to TRUE. If set to FALSE, distances will be calculated from the other nodes towards the reference node instead.

remove_multiple
: Should multiple edges be merged into one. Defaults to TRUE.

remove_loops
: Should loop edges be removed. Defaults to TRUE.

protect
: Nodes to be protected from being removed, no matter if they are a pseudo node or not. Can be given as a numeric vector containing node indices or a character vector containing node names. Can also be a set of geospatial features as object of class sf or sfc. In that case, for each of these features its nearest node in the network will be protected. Defaults to NULL, meaning that none of the nodes is protected.

require_equal
: Should nodes only be removed when the attribute values of their incident edges are equal? Defaults to FALSE. If TRUE, only pseudo nodes that have incident edges with equal attribute values are removed. May also be given as a vector of attribute names. In that case only those attributes are checked for equality. Equality tests are evaluated using the == operator.

subset_by
: Whether to create subgraphs based on nodes or edges.

Details

It 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 the group of nodes will be used as geometry of the contracted node. If edge 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_neighborhood(): Limit a network to the spatial neighborhood of a specific node. ... is forwarded to node_distance_from (if from is TRUE) or node_distance_to (if from is FALSE). 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 with type = 'shortest'. 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(): Remove loop edges and/or merges multiple edges into a single edge. Multiple edges are edges that have the same source and target nodes (in directed networks) or edges that are incident to the same nodes (in undirected networks). 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 each interior point that is equal to any other interior or boundary point in the edges table. Interior points in this sense are those points that are included in their linestring geometry feature but are not endpoints of it, while boundary points are the endpoints of the linestrings. The network is reconstructed after subdivision such that edges are connected at the points of subdivision. Returns a morphed_sfnetwork containing a single element of class sfnetwork. This morpher requires edges to be spatially explicit and nodes to be spatially unique (i.e. not more than one node at the same spatial location).
- 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`.

See Also

The vignette on spatial morphers.

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") %>%
 mutate(weight = edge_length()) %>%
 morph(to_spatial_shortest_paths, from = 1, to = 10) %>%
 mutate(in_paths = TRUE) %>%
 unmorph()

# Lasting changes with convert.
net %>%
 activate("edges") %>%
 mutate(weight = edge_length()) %>%
 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 `any(predicate(x[i], y[j]))` is `TRUE`. Hence, in the case of using `node_intersects`, element i in the returned vector is `TRUE` when node i intersects with any of the features given 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, ...)
```

Arguments

y	The geospatial features to test the nodes against, either as an object of class `sf` or `sfc`.
...	Arguments passed on to the corresponding spatial predicate function of sf. See `geos_binary_pred`.

Details

See `geos_binary_pred` for details on each spatial predicate. 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)
```

---

st_network_bbox *Get the bounding box of a spatial network*

Description

A spatial network specific bounding box extractor, 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)

# Create a network.
node1 = st_point(c(8, 51))
node2 = st_point(c(7, 51.5))
node3 = st_point(c(8, 52))
node4 = st_point(c(9, 51))
edge1 = st_sfc(st_linestring(c(node1, node2, node3)))

nodes = st_as_sf(c(st_sfc(node1), st_sfc(node3), st_sfc(node4)))
edges = st_as_sf(edge1)
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.
oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1), mfrow = c(1,2))
plot(net, lwd = 2, cex = 4, main = "Element bounding boxes")
plot(st_as_sfc(node_bbox), border = "red", lty = 2, lwd = 4, add = TRUE)
plot(st_as_sfc(edge_bbox), border = "blue", lty = 2, lwd = 4, add = TRUE)
plot(net, lwd = 2, cex = 4, main = "Network bounding box")
plot(st_as_sfc(net_bbox), border = "red", lty = 2, lwd = 4, add = TRUE)
par(oldpar)
```

---

st_network_blend *Blend geospatial points into a spatial network*

Description

Blending a point into a network is the combined process of first snapping the given point to its nearest point on its nearest edge in the network, subsequently splitting that edge at the location of the snapped point, and finally adding the snapped point as node to the network. If the location of the snapped point is already a node in the network, the attributes of the point (if any) will be joined to that node.

Usage

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

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.

Details

There are two important details to be aware of. Firstly: when the snap locations of multiple points are equal, only the first of these points is blended into the network. By arranging y before blending you can influence which (type of) point is given priority in such cases. Secondly: when the snap location of a point intersects with multiple edges, it is only blended into the first of these edges. You might want to run the to_spatial_subdivision morpher after blending, such that intersecting but unconnected edges get connected.

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 the line.

Examples

```
library(sf, quietly = TRUE)

# Create a network and a set of points to blend.
n11 = st_point(c(0,0))
n12 = st_point(c(1,1))
e1 = st_sfc(st_linestring(c(n11, n12)), crs = 3857)

n21 = n12
n22 = st_point(c(0,2))
e2 = st_sfc(st_linestring(c(n21, n22)), crs = 3857)

n31 = n22
n32 = st_point(c(-1,1))
e3 = st_sfc(st_linestring(c(n31, n32)), crs = 3857)

net = as_sfnetwork(c(e1,e2,e3))
```

```
pts = net %>%
  st_bbox() %>%
  st_as_sfc() %>%
  st_sample(10, type = "random") %>%
  st_set_crs(3857) %>%
  st_cast('POINT')

# Blend points into the network.
# --> By default tolerance is set to Inf
# --> Meaning that all points get blended
b1 = st_network_blend(net, pts)
b1

# Blend points with a tolerance.
tol = units::set_units(0.2, "m")
b2 = st_network_blend(net, pts, tolerance = tol)
b2

## Plot results.
# Initial network and points.
oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1), mfrow = c(1,3))
plot(net, cex = 2, main = "Network + set of points")
plot(pts, cex = 2, col = "red", pch = 20, add = TRUE)

# Blend with no tolerance
plot(b1, cex = 2, main = "Blend with tolerance = Inf")
plot(pts, cex = 2, col = "red", pch = 20, add = TRUE)

# Blend with tolerance.
within = st_is_within_distance(pts, st_geometry(net, "edges"), tol)
pts_within = pts[lengths(within) > 0]
plot(b2, cex = 2, main = "Blend with tolerance = 0.2 m")
plot(pts, cex = 2, col = "grey", pch = 20, add = TRUE)
plot(pts_within, cex = 2, col = "red", pch = 20, add = TRUE)
par(oldpar)
```

`st_network_cost` *Compute a cost matrix of a spatial network*

Description

Wrapper around `distances` to calculate costs of pairwise shortest paths between points in a spatial network. It allows to provide any set of geospatial point as `from` and `to` arguments. If such a geospatial point is not equal to a node in the network, it will be snapped to its nearest node before calculating costs.

Usage

```
st_network_cost(
  x,
  from = igraph::V(x),
  to = igraph::V(x),
  weights = NULL,
  direction = "out",
  Inf_as_NaN = FALSE,
  ...
)
```

Arguments

`x`	An object of class `sfnetwork`.
`from`	The (set of) geospatial point(s) from which the shortest paths will be calculated. Can be an object of class `sf` or `sfc`. Alternatively it can be a numeric vector containing the indices of the nodes from which the shortest paths will be calculated, or a character vector containing the names of the nodes from which the shortest paths will be calculated. By default, all nodes in the network are included.
`to`	The (set of) geospatial point(s) to which the shortest paths will be calculated. Can be an object of class `sf` or `sfc`. Alternatively it can be a numeric vector containing the indices of the nodes to which the shortest paths will be calculated, or a character vector containing the names of the nodes to which the shortest paths will be calculated. Duplicated values will be removed before calculating the cost matrix. By default, all nodes in the network are included.
`weights`	The edge weights to be used in the shortest path calculation. Can be a numeric vector giving edge weights, or a column name referring to an attribute column in the edges table containing those weights. If set to `NULL`, the values of a column named `weight` in the edges table will be used automatically, as long as this column is present. If not, the geographic edge lengths will be calculated internally and used as weights. If set to `NA`, no weights are used, even if the edges have a `weight` column.
`direction`	The direction of travel. Defaults to `'out'`, meaning that the direction given by the network is followed and costs are calculated from the points given as argument `from`. May be set to `'in'`, meaning that the opposite direction is followed an costs are calculated 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`.
`...`	Arguments passed on to `distances`. Argument `mode` is ignored. Use `direction` instead.

Details

Spatial features provided to the `from` and/or `to` argument don't necessarily have to be points. Internally, the nearest node to each feature is found by calling `st_nearest_feature`, so any feature with a geometry type that is accepted by that function can be provided as from and/or to argument.

When directly providing integer node indices or character node names to the from and/or to argument, keep the following in mind. A node index should correspond to a row-number of the nodes table of the network. A node name should correspond to a value of a column in the nodes table named name. This column should contain character values without duplicates.

For more details on the wrapped function from igraph see the distances documentation page.

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

Examples

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

# Create a network with edge lengths as weights.
# These weights will be used automatically in shortest paths calculation.
net = as_sfnetwork(roxel, directed = FALSE) %>%
  st_transform(3035) %>%
  activate("edges") %>%
  mutate(weight = edge_length())

# Providing node indices.
st_network_cost(net, from = c(495, 121), to = c(495, 121))

# Providing nodes as spatial points.
# Points that don't equal a node will be snapped to their nearest node.
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))

# Using another column for weights.
net %>%
  activate("edges") %>%
  mutate(foo = runif(n(), min = 0, max = 1)) %>%
  st_network_cost(c(p1, p2), c(p1, p2), weights = "foo")

# 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_join — *Join two spatial networks based on equality of node geometries*

Description

A spatial network specific join function which makes a spatial full join on the geometries of the nodes data, based on the st_equals spatial predicate. 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.

Examples

```
library(sf, quietly = TRUE)

node1 = st_point(c(0, 0))
node2 = st_point(c(1, 0))
node3 = st_point(c(1,1))
node4 = st_point(c(0,1))
edge1 = st_sfc(st_linestring(c(node1, node2)))
edge2 = st_sfc(st_linestring(c(node2, node3)))
edge3 = st_sfc(st_linestring(c(node3, node4)))

net1 = as_sfnetwork(c(edge1, edge2))
net2 = as_sfnetwork(c(edge2, edge3))

joined = st_network_join(net1, net2)
joined

## Plot results.
oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1), mfrow = c(1,2))
plot(net1, pch = 15, cex = 2, lwd = 4)
plot(net2, col = "red", pch = 18, cex = 2, lty = 3, lwd = 4, add = TRUE)
```

```
plot(joined, cex = 2, lwd = 4)
par(oldpar)
```

---

st_network_paths *Paths between points in geographical space*

Description

Combined wrapper around shortest_paths, all_shortest_paths and all_simple_paths from igraph, allowing to provide any geospatial point as from argument and any set of geospatial points as to argument. If such a geospatial point is not equal to a node in the network, it will be snapped to its nearest node before calculating the shortest or simple paths.

Usage

```
st_network_paths(
  x,
  from,
  to = igraph::V(x),
  weights = NULL,
  type = "shortest",
  use_names = TRUE,
  ...
)
```

Arguments

x	An object of class sfnetwork.
from	The geospatial point from which the paths will be calculated. Can be an object an object of class sf or sfc, containing a single feature. When multiple features are given, only the first one is used. Alternatively, it can be an integer, referring to the index of the node from which the paths will be calculated, or a character, referring to the name of the node from which the paths will be calculated.
to	The (set of) geospatial point(s) to which the paths will be calculated. Can be an object of class sf or sfc. Alternatively it can be a numeric vector containing the indices of the nodes to which the paths will be calculated, or a character vector containing the names of the nodes to which the paths will be calculated. By default, all nodes in the network are included.
weights	The edge weights to be used in the shortest path calculation. Can be a numeric vector giving edge weights, or a column name referring to an attribute column in the edges table containing those weights. If set to NULL, the values of a column named weight in the edges table will be used automatically, as long as this column is present. If not, the geographic edge lengths will be calculated internally and used as weights. If set to NA, no weights are used, even if the edges have a weight column. Ignored when type = 'all_simple'.

`type` Character defining which type of path calculation should be performed. If set to `'shortest'` paths are calculated using `shortest_paths`, if set to `'all_shortest'` paths are calculated using `all_shortest_paths`, if set to `'all_simple'` paths are calculated using `all_simple_paths`. Defaults to `'shortest'`.

`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 `TRUE`. Ignored when the nodes table does not have a column named `name`.

`...` Arguments passed on to the corresponding `igraph` or `igraph` function. Arguments `predecessors` and `inbound.edges` are ignored.

Details

Spatial features provided to the `from` and/or `to` argument don't necessarily have to be points. Internally, the nearest node to each feature is found by calling `st_nearest_feature`, so any feature with a geometry type that is accepted by that function can be provided as `from` and/or `to` argument.

When directly providing integer node indices or character node names to the `from` and/or `to` argument, keep the following in mind. A node index should correspond to a row-number of the nodes table of the network. A node name should correspond to a value of a column in the nodes table named `name`. This column should contain character values without duplicates.

For more details on the wrapped functions from `igraph` see the `shortest_paths` or `all_simple_paths` documentation pages.

Value

An object of class `tbl_df` with one row per returned path. Depending on the setting of the `type` argument, columns can be `node_paths` (a list column with for each path the ordered indices of nodes present in that path) and `edge_paths` (a list column with for each path the ordered indices of edges present in that path). `'all_shortest'` and `'all_simple'` return only `node_paths`, while `'shortest'` returns both.

See Also

`st_network_cost`

Examples

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

# Create a network with edge lengths as weights.
# These weights will be used automatically in shortest paths calculation.
net = as_sfnetwork(roxel, directed = FALSE) %>%
  st_transform(3035) %>%
  activate("edges") %>%
  mutate(weight = edge_length())

# Providing node indices.
paths = st_network_paths(net, from = 495, to = 121)
paths
```

```
node_path = paths %>%
  slice(1) %>%
  pull(node_paths) %>%
  unlist()
node_path

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1))
plot(net, col = "grey")
plot(slice(activate(net, "nodes"), node_path), col = "red", add = TRUE)
par(oldpar)

# Providing nodes as spatial points.
# Points that don't equal a node will be snapped to their nearest node.
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

node_path = paths %>%
  slice(1) %>%
  pull(node_paths) %>%
  unlist()
node_path

oldpar = par(no.readonly = TRUE)
par(mar = c(1,1,1,1))
plot(net, col = "grey")
plot(c(p1, p2), col = "black", pch = 8, add = TRUE)
plot(slice(activate(net, "nodes"), node_path), col = "red", add = TRUE)
par(oldpar)

# Using another column for weights.
net %>%
  activate("edges") %>%
  mutate(foo = runif(n(), min = 0, max = 1)) %>%
  st_network_paths(p1, p2, weights = "foo")

# Obtaining all simple paths between two nodes.
# Beware, this function can take long when:
# --> Providing a lot of 'to' nodes.
# --> The network is large and dense.
net = as_sfnetwork(roxel, directed = TRUE)
st_network_paths(net, from = 1, to = 12, type = "all_simple")

# Obtaining all shortest paths between two nodes.
# Not using edge weights.
# Hence, a shortest path is the paths with the least number of edges.
st_network_paths(net, from = 5, to = 1, weights = NA, type = "all_shortest")
```

Index

∗ **datasets**
roxel, 9

all_shortest_paths, *34, 35*
all_simple_paths, *34, 35*
arrange, *24*
as.linnet, 2
as_s2_geography.sfnetwork *(*s2*)*, 10
as_sfnetwork, *3*, 3, *33*
as_tbl_graph, *3*
as_tibble, *5*, 5
autoplot, *6*, 6

bbox, *13*, *27*
bind_rows, *33*

convert, *21*
crystallise, *21*

data.frame, *15*
distances, *30–32*

edge_azimuth *(*spatial_edge_measures*)*, 17
edge_circuity *(*spatial_edge_measures*)*, 17
edge_contains *(*spatial_edge_predicates*)*, 19
edge_contains_properly *(*spatial_edge_predicates*)*, 19
edge_covers *(*spatial_edge_predicates*)*, 19
edge_crosses *(*spatial_edge_predicates*)*, 19
edge_displacement *(*spatial_edge_measures*)*, 17
edge_equals *(*spatial_edge_predicates*)*, 19
edge_intersects *(*spatial_edge_predicates*)*, 19
edge_is_covered_by *(*spatial_edge_predicates*)*, 19
edge_is_disjoint *(*spatial_edge_predicates*)*, 19
edge_is_within *(*spatial_edge_predicates*)*, 19
edge_is_within_distance *(*spatial_edge_predicates*)*, 19
edge_length *(*spatial_edge_measures*)*, 17
edge_overlaps *(*spatial_edge_predicates*)*, 19
edge_touches *(*spatial_edge_predicates*)*, 19

filter, *7*, *18–20*, *25, 26*

geos_binary_pred, *20*, *26*
ggplot, *6*
graph_join, *33*
group_by, *23*

igraph, *32*, *34, 35*
is.sfnetwork, 6

linnet, *2, 3*

morph, *21*, *23*
morphers, *21*
mutate, *7*, *18–20*, *25, 26*

node_coordinates, 7
node_distance_from, *24*
node_distance_to, *24*, *26*
node_equals *(*spatial_node_predicates*)*, 25
node_intersects *(*spatial_node_predicates*)*, 25
node_is_covered_by *(*spatial_node_predicates*)*, 25
node_is_disjoint *(*spatial_node_predicates*)*, 25
node_is_within *(*spatial_node_predicates*)*, 25

node_is_within_distance (spatial_node_predicates), 25
node_M (node_coordinates), 7
node_touches (spatial_node_predicates), 25
node_X (node_coordinates), 7
node_Y (node_coordinates), 7
node_Z (node_coordinates), 7

plot.sf, *8*
plot.sfnetwork, 8

roxel, 9

s2, 10
sf, *5*, *9*, *10*, 10, *13–16*, *20*, *22–24*, *26*, *29*, *31*, *34*
sf_attr, 17
sfc, *20*, *22*, *23*, *26*, *29*, *31*, *34*
sfg, *13*
sfnetwork, *2*, *3*, *5*, *6*, *8*, *10*, *12*, *13*, 14, *17*, *22–25*, *27*, *29*, *31*, *33*, *34*
shortest_paths, *34*, *35*
spatial_edge_measures, 17
spatial_edge_predicates, 19
spatial_morphers, 21
spatial_node_predicates, 25
st_agr.sfnetwork (sf), 10
st_agr<-.sfnetwork (sf), 10
st_area.sfnetwork (sf), 10
st_as_s2.sfnetwork (sf), 10
st_as_sf, *3*, *13*, *15*, *16*, *24*
st_as_sf.sfnetwork (sf), 10
st_bbox, *27*
st_bbox.sfnetwork (sf), 10
st_coordinates.sfnetwork (sf), 10
st_crop, *13*
st_crop.morphed_sfnetwork (sf), 10
st_crop.sfnetwork (sf), 10
st_crs.sfnetwork (sf), 10
st_crs<-.sfnetwork (sf), 10
st_difference, *13*
st_difference.morphed_sfnetwork (sf), 10
st_difference.sfnetwork (sf), 10
st_distance, *18*
st_drop_geometry.sfnetwork (sf), 10
st_equals, *33*
st_filter, *13*, *24*
st_filter.morphed_sfnetwork (sf), 10
st_filter.sfnetwork (sf), 10
st_geod_azimuth, *18*
st_geometry.sfnetwork (sf), 10
st_geometry<-.sfnetwork (sf), 10
st_intersection, *13*
st_intersection.morphed_sfnetwork (sf), 10
st_intersection.sfnetwork (sf), 10
st_intersects.sfnetwork (sf), 10
st_is.sfnetwork (sf), 10
st_is_valid.sfnetwork (sf), 10
st_join, *13*
st_join.morphed_sfnetwork (sf), 10
st_join.sfnetwork (sf), 10
st_length, *18*
st_m_range.sfnetwork (sf), 10
st_nearest_feature, *31*, *35*
st_nearest_points.sfnetwork (sf), 10
st_network_bbox, 27
st_network_blend, 28
st_network_cost, 30, *35*
st_network_join, 33
st_network_paths, *24*, *32*, 34
st_normalize.sfnetwork (sf), 10
st_precision, *13*
st_precision.sfnetwork (sf), 10
st_reverse.sfnetwork (sf), 10
st_sample.sfnetwork (sf), 10
st_set_precision.sfnetwork (sf), 10
st_shift_longitude.sfnetwork (sf), 10
st_simplify.sfnetwork (sf), 10
st_transform, *25*
st_transform.sfnetwork (sf), 10
st_wrap_dateline.sfnetwork (sf), 10
st_z_range.sfnetwork (sf), 10
st_zm.sfnetwork (sf), 10

tbl_df, *15*, *21*, *35*
tbl_graph, *14*, *23*
tibble, *5*
tidygraph, *17*
to_directed, *24*
to_spatial_contracted (spatial_morphers), 21
to_spatial_directed (spatial_morphers), 21
to_spatial_explicit (spatial_morphers), 21

to_spatial_neighborhood
(spatial_morphers), 21
to_spatial_shortest_paths
(spatial_morphers), 21
to_spatial_simple (spatial_morphers), 21
to_spatial_smooth (spatial_morphers), 21
to_spatial_subdivision, *29*
to_spatial_subdivision
(spatial_morphers), 21
to_spatial_subset (spatial_morphers), 21
to_spatial_transformed
(spatial_morphers), 21
to_subgraph, *24*

units, *29*

with_graph, *7*, *18*, *20*, *26*