Package ‘tidygraph’

May 12, 2020

**Type**  Package  

**Title**  A Tidy API for Graph Manipulation  

**Version**  1.2.0  

**Maintainer**  Thomas Lin Pedersen <thomasp85@gmail.com>  

**Description**  A graph, while not ``tidy'' in itself, can be thought of as two tidy data frames describing node and edge data respectively. 'tidygraph' provides an approach to manipulate these two virtual data frames using the API defined in the 'dplyr' package, as well as provides tidy interfaces to a lot of common graph algorithms.  

**License**  MIT + file LICENSE  

**Encoding**  UTF-8  

**LazyData**  true  

**RoxygenNote**  7.1.0  

**Imports**  tibble, dplyr (>= 0.8.5), igraph, magrittr, utils, rlang, R6, Rcpp, tools, stats, tidyr, pillar  

**URL**  https://tidygraph.data-imaginist.com,  

https://github.com/thomasp85/tidygraph  

**BugReports**  https://github.com/thomasp85/tidygraph/issues  

**LinkingTo**  Rcpp  

**Suggests**  network, data.tree, ape, graph, methods, testthat, covr, seriation, netrankr, influenceR, NetSwan  

**NeedsCompilation**  yes  

**Author**  Thomas Lin Pedersen [cre, aut]  

(http://orcid.org/0000-0002-5147-4711)  

**Repository**  CRAN  

**Date/Publication**  2020-05-12 07:30:03 UTC
activate

Determine the context of subsequent manipulations

As a tbl_graph can be considered as a collection of two linked tables it is necessary to specify which table is referenced during manipulations. The activate verb does just that and needs affects all subsequent manipulations until a new table is activated. active is a simple query function to get the currently active context. In addition to the use of activate it is also possible to activate nodes or edges as part of the piping using the %N>% and %E>% pipes respectively. Do note that this approach somewhat obscures what is going on and is thus only recommended for quick, one-line, fixes in interactive use.
Usage

activate(.data, what)
active(x)

lhs %N>% rhs
lhs %E>% rhs

Arguments

.data, x, lhs  A tbl_graph or a grouped_tbl_graph
what          What should get activated? Possible values are nodes or edges.
rhs          A function to pipe into

Value

A tbl_graph

Note

Activate will ungroup a grouped_tbl_graph.

Examples

gr <- create_complete(5) %>%
  activate(nodes) %>%
  mutate(class = sample(c('a', 'b'), 5, TRUE)) %>%
  activate(edges) %>%
  arrange(from)

# The above could be achieved using the special pipes as well
gr <- create_complete(5) %N>%
  mutate(class = sample(c('a', 'b'), 5, TRUE)) %E>%
  arrange(from)
# But as you can see it obscures what part of the graph is being targeted
Description

The tbl_graph class is a thin wrapper around an igraph object that provides methods for manipulating the graph using the tidy API. As it is just a subclass of igraph every igraph method will work as expected. A grouped_tbl_graph is the equivalent of a grouped_df where either the nodes or the edges has been grouped. The grouped_tbl_graph is not constructed directly but by using the `group_by()` verb. After creation of a tbl_graph the nodes are activated by default. The context can be changed using the `activate()` verb and affects all subsequent operations. Changing context automatically drops any grouping. The current active context can always be extracted with `as_tibble()`, which drops the graph structure and just returns a tbl_df or a grouped_df depending on the state of the tbl_graph. The returned context can be overridden by using the active argument in `as_tibble()`.

Usage

```r
## S3 method for class 'data.frame'
as_tbl_graph(x, directed = TRUE, ...)  
## S3 method for class 'Node'
as_tbl_graph(x, directed = TRUE, mode = "out", ...)  
## S3 method for class 'dendrogram'
as_tbl_graph(x, directed = TRUE, mode = "out", ...)  
## S3 method for class 'graphNEL'
as_tbl_graph(x, ...)  
## S3 method for class 'graphAM'
as_tbl_graph(x, ...)  
## S3 method for class 'graphBAM'
as_tbl_graph(x, ...)  
## S3 method for class 'hclust'
as_tbl_graph(x, directed = TRUE, mode = "out", ...)  
## S3 method for class 'igraph'
as_tbl_graph(x, ...)  
## S3 method for class 'list'
as_tbl_graph(x, directed = TRUE, node_key = "name", ...)  
## S3 method for class 'matrix'
as_tbl_graph(x, directed = TRUE, ...)  
## S3 method for class 'network'
as_tbl_graph(x, ...)  
## S3 method for class 'phylo'
```
as_tbl_graph.data.frame

as_tbl_graph(x, directed = NULL, ...)

## S3 method for class 'evonet'
as_tbl_graph(x, directed = TRUE, ...)

tbl_graph(nodes = NULL, edges = NULL, directed = TRUE, node_key = "name")
as_tbl_graph(x, ...)

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

is.tbl_graph(x)

Arguments

x An object convertible to a tbl_graph
directed Should the constructed graph be directed (defaults to TRUE)
... Arguments passed on to the conversion function
node In case directed = TRUE should the edge direction be away from node or towards. Possible values are "out" (default) or "in".
node_key The name of the column in nodes 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.
nodes A data.frame containing information about the nodes in the graph. If edges$to and/or edges$from are characters then they will be matched to the column named according to node_key in nodes, if it exists. If not, they will be matched to the first column.
edges A data.frame containing information about the edges in the graph. The terminal nodes of each edge must either be encoded in a to and from column, or in the two first columns, as integers. These integers refer to nodes index.

Details

Constructors are provided for most data structures that resembles networks. If a class provides an igraph::as.igraph() method it is automatically supported.

Value

A tbl_graph object

Methods (by generic)

- as_tbl_graph: Method for edge table and set membership table
- as_tbl_graph: Method to deal with Node objects from the data.tree package
- as_tbl_graph: Method for dendrogram objects
• as_tbl_graph: Method for handling graphNEL objects from the graph package (on Bioconductor)
• as_tbl_graph: Method for handling graphAM objects from the graph package (on Bioconductor)
• as_tbl_graph: Method for handling graphBAM objects from the graph package (on Bioconductor)
• as_tbl_graph: Method for hclust objects
• as_tbl_graph: Method for igraph object. Simply subclasses the object into a tbl_graph
• as_tbl_graph: Method for adjacency lists and lists of node and edge tables
• as_tbl_graph: Method for edgelist, adjacency and incidence matrices
• as_tbl_graph: Method to handle network objects from the network package. Requires this packages to work.
• as_tbl_graph: Method for handling phylo objects from the ape package
• as_tbl_graph: Method for handling evonet objects from the ape package
• as_tbl_graph: Default method. tries to call igraph::as.igraph() on the input.

Examples

rstat_nodes <- data.frame(name = c("Hadley", "David", "Romain", "Julia"))
rstat_edges <- data.frame(from = c(1, 1, 1, 2, 3, 3, 4, 4, 4),
                          to = c(2, 3, 4, 1, 1, 2, 1, 2, 3))
tbl_graph(nodes = rstat_nodes, edges = rstat_edges)

bind_graphs Add graphs, nodes, or edges to a tbl_graph

Description

These functions are tbl_graph pendants to dplyr::bind_rows() that allows you to grow your tbl_graph by adding rows to either the nodes data, the edges data, or both. As with bind_rows() columns are matched by name and are automatically filled with NA if the column doesn’t exist in some instances. In the case of bind_graphs() the graphs are automatically converted to tbl_graph objects prior to binding. The edges in each graph will continue to reference the nodes in the graph where they originated, meaning that their terminal node indexes will be shifted to match the new index of the node in the combined graph. This means the bind_graphs() always result in a disconnected graph. See graph_join() for merging graphs on common nodes.

Usage

bind_graphs(.data, ...)
bind_nodes(.data, ...)
bind_edges(.data, ..., node_key = "name")
centrality

Arguments

.data A tbl_graph, or a list of tbl_graph objects (for bind_graphs()).

... In case of bind_nodes() and bind_edges() data.frames to add. In the case of
bind_graphs() objects that are convertible to tbl_graph using as_tbl_graph().

node_key The name of the column in nodes 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.

Value

A tbl_graph containing the new data

Examples

graph <- create_notable('bull')
new_graph <- create_notable('housex')

# Add nodes
graph %>% bind_nodes(data.frame(new = 1:4))

# Add edges
graph %>% bind_edges(data.frame(from = 1, to = 4:5))

# Add graphs
graph %>% bind_graphs(new_graph)

centrality Calculate node and edge centrality

Description

The centrality of a node measures the importance of node in the network. As the concept of
importance is ill-defined and dependent on the network and the questions under consideration, many
centrality measures exist. tidygraph provides a consistent set of wrappers for all the centrality
measures implemented in igraph for use inside dplyr::mutate() and other relevant verbs. All
functions provided by tidygraph have a consistent naming scheme and automatically calls the
function on the graph, returning a vector with measures ready to be added to the node data. Further
tidygraph provides access to the netrankr engine for centrality calculations and define a number
of centrality measures based on that, as well as provide a manual mode for specifying more-or-less
any centrality score.

Usage

centrality_alpha(
weights = NULL,
alpha = 1,
exo = 1,
  tol = 1e-07,
  loops = FALSE
)

centrality_authority(
  weights = NULL,
  scale = TRUE,
  options = igraph::arpack_defaults
)

centrality_betweenness(
  weights = NULL,
  directed = TRUE,
  cutoff = NULL,
  nobigint = TRUE,
  normalized = FALSE
)

centrality_power(exponent = 1, rescale = FALSE, tol = 1e-07, loops = FALSE)

centrality_closeness(
  weights = NULL,
  mode = "out",
  normalized = FALSE,
  cutoff = NULL
)

centrality_eigen(
  weights = NULL,
  directed = FALSE,
  scale = TRUE,
  options = igraph::arpack_defaults
)

centrality_hub(weights = NULL, scale = TRUE, options = igraph::arpack_defaults)

centrality_pagerank(
  weights = NULL,
  directed = TRUE,
  damping = 0.85,
  personalized = NULL
)

centrality_subgraph(loops = FALSE)

centrality_degree(
  weights = NULL,
centrality

mode = "out",
loops = TRUE,
normalized = FALSE
)

centrality_edge_betweenness(weights = NULL, directed = TRUE, cutoff = NULL)

centrality_manual(relation = "dist_sp", aggregation = "sum", ...)

centrality_closeness_harmonic()

centrality_closeness_residual()

centrality_closeness_generalised(alpha)

centrality_integration()

centrality_communicability()

centrality_communicability_odd()

centrality_communicability_even()

centrality_subgraph_odd()

centrality_subgraph_even()

centrality_katz(alpha = NULL)

centrality_betweenness_network(netflowmode = "raw")

centrality_betweenness_current()

centrality_betweenness_communicability()

centrality_betweenness_rsp_simple(rspxparam = 1)

centrality_betweenness_rsp_net(rspxparam = 1)

centrality_information()

centrality_decay(alpha = 1)

centrality_random_walk()

centrality_expected()
Arguments

weights The weight of the edges to use for the calculation. Will be evaluated in the context of the edge data.
alpha Relative importance of endogenous vs exogenous factors (centrality_alpha), the exponent to the power transformation of the distance metric (centrality_closeness_generalised), the base of power transformation (centrality_decay), or the attenuation factor (centrality_katz)
exo The exogenous factors of the nodes. Either a scalar or a number number for each node. Evaluated in the context of the node data.
tol Tolerance for near-singularities during matrix inversion
loops Should loops be included in the calculation
scale Should the output be scaled between 0 and 1
options Settings passed on to igraph::arpack()
directed Should direction of edges be used for the calculations
cutoff maximum path length to use during calculations
nobigint Should big integers be avoided during calculations
normalized Should the output be normalized
exponent The decay rate for the Bonacich power centrality
rescale Should the output be scaled to sum up to 1
mode How should edges be followed. Ignored for undirected graphs
damping The damping factor of the page rank algorithm
personalized The probability of jumping to a node when abandoning a random walk. Evaluated in the context of the node data.
relation The indirect relation measure type to be used in netrankr::indirect_relations
aggregation The aggregation type to use on the indirect relations to be used in netrankr::aggregate_positions
... Arguments to pass on to netrankr::indirect_relations
netflowmode The return type of the network flow distance, either 'raw' or 'frac'
rspxparam inverse temperature parameter

Value

A numeric vector giving the centrality measure of each node.

Functions

- centrality_alpha: Wrapper for igraph::alpha_centrality()
- centrality_authority: Wrapper for igraph::authority_score()
- centrality_betweenness: Wrapper for igraph::betweenness() and igraph::estimate_betweenness()
- centrality_power: Wrapper for igraph::power_centrality()
- centrality_closeness: Wrapper for igraph::closeness() and igraph::estimate_closeness()
- centrality_eigen: Wrapper for igraph::eigen_centrality()
- centrality_hub: Wrapper for `igraph::hub_score()`
- centrality_pagerank: Wrapper for `igraph::page_rank()`
- centrality_subgraph: Wrapper for `igraph::subgraph_centRALity()`
- centrality_degree: Wrapper for `igraph::degree()` and `igraph::strength()`
- centrality_edge_betweenness: Wrapper for `igraph::edge_betweenness()`
- centrality_manual: Manually specify your centrality score using the netrankr framework (netrankr)
- centrality_closeness_harmonic: centrality based on inverse shortest path (netrankr)
- centrality_closeness_residual: centrality based on 2-to-the-power-of negative shortest path (netrankr)
- centrality_closeness_generalised: centrality based on alpha-to-the-power-of negative shortest path (netrankr)
- centrality_integration: centrality based on $1 - (x-1)/\max(x)$ transformation of shortest path (netrankr)
- centrality_communicability: centrality an exponential tranformation of walk counts (netrankr)
- centrality_communicability_odd: centrality an exponential transformation of odd walk counts (netrankr)
- centrality_communicability_even: centrality an exponential transformation of even walk counts (netrankr)
- centrality_subgraph_odd: subgraph centrality based on odd walk counts (netrankr)
- centrality_subgraph_even: subgraph centrality based on even walk counts (netrankr)
- centrality_katz: centrality based on walks penalizing distant nodes (netrankr)
- centrality_betweenness_network: Betweenness centrality based on network flow (netrankr)
- centrality_betweenness_current: Betweenness centrality based on current flow (netrankr)
- centrality_betweenness_communicability: Betweenness centrality based on communicability (netrankr)
- centrality_betweenness_rsp_simple: Betweenness centrality based on simple randomised shortest path dependencies (netrankr)
- centrality_betweenness_rsp_net: Betweenness centrality based on net randomised shortest path dependencies (netrankr)
- centrality_information: centrality based on inverse sum of resistance distance between nodes (netrankr)
- centrality_decay: based on a power transformation of the shortest path (netrankr)
- centrality_random_walk: centrality based on the inverse sum of expected random walk length between nodes (netrankr)
- centrality_expected: Expected centrality ranking based on exact rank probability (netrankr)
Examples

create_notable('bull') %>%
  activate(nodes) %>%
  mutate(importance = centrality_alpha())

# Most centrality measures are for nodes but not all
create_notable('bull') %>%
  activate(edges) %>%
  mutate(importance = centrality_edge_betweenness())

component_games Graph games based on connected components

Description

This set of graph creation algorithms simulate the topology by, in some way, connecting subgraphs. The nature of their algorithm is described in detail at the linked igraph documentation.

Usage

play_blocks(n, size_blocks, p_between, directed = TRUE, loops = FALSE)

play_blocks_hierarchy(n, size_blocks, rho, p_within, p_between)

play_islands(n_islands, size_islands, p_within, m_between)

play_smallworld(
  n_dim,
  dim_size,
  order,
  p_rewire,
  loops = FALSE,
  multiple = FALSE
)

Arguments

n The number of nodes in the graph.
size_blocks The number of vertices in each block
p_between, p_within The probability of edges within and between groups/blocks
directed Should the resulting graph be directed
loops Are loop edges allowed
rho The fraction of vertices per cluster
n_islands The number of densely connected islands
The number of nodes in each island
The number of edges between groups/islands
The dimension and size of the starting lattice
The neighborhood size to create connections from
The rewiring probability of edges
Are multiple edges allowed

A tbl_graph object

• play_blocks: Create graphs by sampling from stochastic block model. See `igraph::sample_sbm()`
• play_blocks_hierarchy: Create graphs by sampling from the hierarchical stochastic block model. See `igraph::sample_hierarchical_sbm()`
• play_islands: Create graphs with fixed size and edge probability of subgraphs as well as fixed edge count between subgraphs. See `igraph::sample_islands()`
• play_smallworld: Create graphs based on the Watts-Strogatz small-world model. See `igraph::sample_smallworld()`

Other graph games: `evolution_games`, `sampling_games`, `type_games`

plot(play_islands(4, 10, 0.7, 3))
Value

Either a tbl_graph (.G()) or a tibble (.N())

Functions

- .G: Get the tbl_graph you’re currently working on
- .N: Get the nodes data from the graph you’re currently working on
- .E: Get the edges data from the graph you’re currently working on

Examples

```r
# Get data from the nodes while computing for the edges
create_notable('bull') %>%
  activate(nodes) %>%
  mutate(centrality = centrality_power()) %>%
  activate(edges) %>%
  mutate(mean_centrality = .N()
    $centrality[from] + .N()
    $centrality[to]) / 2)
```

create_graphs

Create different types of well-defined graphs

Description

These functions creates a long list of different types of well-defined graphs, that is, their structure is not based on any randomisation. All of these functions are shallow wrappers around a range of igraph::make_* functions but returns tbl_graph rather than igraph objects.

Usage

```r
create_ring(n, directed = FALSE, mutual = FALSE)
create_path(n, directed = FALSE, mutual = FALSE)
create_chordal_ring(n, w)
create_de_bruijn(alphabet_size, label_size)
create_empty(n, directed = FALSE)
create_bipartite(n1, n2, directed = FALSE, mode = "out")
create_citation(n)
create_complete(n)
create_notable(name)
```
create_kautz(alphabet_size, label_size)

create_lattice(dim, directed = FALSE, mutual = FALSE, circular = FALSE)

create_star(n, directed = FALSE, mutual = FALSE, mode = "out")

create_tree(n, children, directed = TRUE, mode = "out")

Arguments

- n, n1, n2: The number of nodes in the graph
- directed: Should the graph be directed
- mutual: Should mutual edges be created in case of the graph being directed
- w: A matrix specifying the additional edges in the chordan ring. See `igraph::make_chordal_ring()`
- alphabet_size: The number of unique letters in the alphabet used for the graph
- label_size: The number of characters in each node
- mode: In case of a directed, non-mutual, graph should the edges flow ‘out’ or ‘in’
- name: The name of a notable graph. See a complete list in `igraph::make_graph()`
- dim: The dimensions of the lattice
- circular: Should each dimension in the lattice wrap around
- children: The number of children each node has in the tree (if possible)

Value

A tbl_graph

Functions

- create_ring: Create a simple ring graph
- create_path: Create a simple path
- create_chordal_ring: Create a chordal ring
- create_de_bruijn: Create a de Bruijn graph with the specified alphabet and label size
- create_empty: Create a graph with no edges
- create_bipartite: Create a full bipartite graph
- create_citation: Create a full citation graph
- create_complete: Create a complete graph (a graph where all nodes are connected)
- create_notable: Create a graph based on its name. See `igraph::make_graph()`
- create_kautz: Create a Kautz graph with the specified alphabet and label size
- create_lattice: Create a multidimensional grid of nodes
- create_star: Create a star graph (A single node in the center connected to all other nodes)
- create_tree: Create a tree graph
Examples

### # Create a complete graph with 10 nodes
create_complete(10)

<table>
<thead>
<tr>
<th>edge_types</th>
<th>Querying edge types</th>
</tr>
</thead>
</table>

Description

These functions let the user query whether the edges in a graph is of a specific type. All functions return a logical vector giving whether each edge in the graph corresponds to the specific type.

Usage

```r
edge_is_multiple()
edge_is_loop()
edge_is_mutual()
edge_is_from(from)
edge_is_to(to)
edge_is_between(from, to, ignore_dir = !graph_is_directed())
edge_is_incident(i)
```

Arguments

- `from, to, i` A vector giving node indices
- `ignore_dir` Is both directions of the edge allowed

Value

A logical vector of the same length as the number of edges in the graph

Functions

- `edge_is_multiple`: Query whether each edge has any parallel siblings
- `edge_is_loop`: Query whether each edge is a loop
- `edge_is_mutual`: Query whether each edge has a sibling going in the reverse direction
- `edge_is_from`: Query whether an edge goes from a set of nodes
- `edge_is_to`: Query whether an edge goes to a set of nodes
- `edge_is_between`: Query whether an edge goes between two sets of nodes
- `edge_is_incident`: Query whether an edge goes from or to a set of nodes
Examples

```r
create_star(10, directed = TRUE, mutual = TRUE) %>%
  activate(edges) %>%
  sample_frac(0.7) %>%
  mutate(single_edge = !edge_is_mutual())
```

Description

This games create graphs through different types of evolutionary mechanisms (not necessarily in a biological sense). The nature of their algorithm is described in detail at the linked igraph documentation.

Usage

```r
play_citation_age(
  n,
  growth = 1,
  bins = n/7100,
  p_pref = (1:(bins + 1))^(-3),
  directed = TRUE
)
```

```r
play_forestfire(
  n,
  p_forward,
  p_backward = p_forward,
  growth = 1,
  directed = TRUE
)
```

```r
play_growing(n, growth = 1, directed = TRUE, citation = FALSE)
```

```r
play_barabasi_albert(
  n,
  power,
  growth = 1,
  growth_dist = NULL,
  use_out = FALSE,
  appeal_zero = 1,
  directed = TRUE,
  method = "psumtree"
)
```

```r
play_barabasi_albert_aging(
```
n, power, power_age, growth = 1, growth_dist = NULL, bins = 300, use_out = FALSE, appeal_zero = 1, appeal_zero_age = 0, directed = TRUE, coefficient = 1, coefficient_age = 1, window = NULL)

Arguments

- **n**: The number of nodes in the graph.
- **growth**: The number of edges added at each iteration.
- **bins**: The number of aging bins.
- **p_pref**: The probability that an edge will be made to an age bin.
- **directed**: Should the resulting graph be directed.
- **p_forward, p_backward**: Forward and backward burning probability.
- **citation**: Should a citation graph be created.
- **power**: The power of the preferential attachment.
- **growth_dist**: The distribution of the number of added edges at each iteration.
- **use_out**: Should outbound edges be used for calculating citation probability.
- **appeal_zero**: The appeal value for unconnected nodes.
- **method**: The algorithm to use for graph creation. Either 'psumtree', 'psumtree-multiple', or 'bag'.
- **power_age**: The aging exponent.
- **appeal_zero_age**: The appeal value of nodes without age.
- **coefficient**: The coefficient of the degree dependent part of attractiveness.
- **coefficient_age**: The coefficient of the age dependent part of attractiveness.
- **window**: The aging window to take into account when calculating the preferential attraction.

Value

A tbl_graph object.
### Functions

- **play_citation_age**: Create citation graphs based on a specific age link probability. See `igraph::sample_last_cit()`
- **play_forestfire**: Create graphs by simulating the spread of fire in a forest. See `igraph::sample_forestfire()`
- **play_growing**: Create graphs by adding a fixed number of edges at each iteration. See `igraph::sample_growing()`
- **play_barabasi_albert**: Create graphs based on the Barabasi-Alberts preferential attachment model. See `igraph::sample_pa()`
- **play_barabasi_albert_aging**: Create graphs based on the Barabasi-Alberts preferential attachment model, incorporating node age preference. See `igraph::sample_pa_age()`.

### See Also

- `play_traits()` and `play_citation_type()` for an evolutionary algorithm based on different node types
- Other graph games: `component_games`, `sampling_games`, `type_games`

### Examples

```r
plot(play_forestfire(50, 0.5))
```

---

**graph_join**

Join graphs on common nodes

---

**Description**

This graph-specific join method makes a full join on the nodes data and updates the edges in the joining graph so they match the new indexes of the nodes in the resulting graph. Node and edge data is combined using `dplyr::bind_rows()` semantic, meaning that data is matched by column name and filled with `NA` if it is missing in either of the graphs.

**Usage**

```r
graph_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ...)
```

**Arguments**

- **x**: A `tbl_graph`
- **y**: An object convertible to a `tbl_graph` using `as_tbl_graph()`
- **by**: A character vector of variables to join by.

If `NULL`, the default, `_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly.
To join by different variables on \( x \) and \( y \), use a named vector. For example, \( \text{by} = c("a" = "b") \) will match \( x$a \) to \( y$b \).

To join by multiple variables, use a vector with length > 1. For example, \( \text{by} = c("a","b") \) will match \( x$a \) to \( y$a \) and \( x$b \) to \( y$b \). Use a named vector to match different variables in \( x \) and \( y \). For example, \( \text{by} = c("a" = "b", "c" = "d") \) will match \( x$a \) to \( y$b \) and \( x$c \) to \( y$d \).

To perform a cross-join, generating all combinations of \( x \) and \( y \), use \( \text{by} = \text{character()} \).

**copy**  
If \( x \) and \( y \) are not from the same data source, and \( \text{copy} \) is \( \text{TRUE} \), then \( y \) will be copied into the same src as \( x \). This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

**suffix**  
If there are non-joined duplicate variables in \( x \) and \( y \), these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.

**...**  
Other parameters passed onto methods.

**Value**  
A tbl_graph containing the merged graph

**Examples**

```r
gr1 <- create_notable('bull') %>%
  activate(nodes) %>%
  mutate(name = letters[1:5])

gr2 <- create_ring(10) %>%
  activate(nodes) %>%
  mutate(name = letters[4:13])

gr1 %>% graph_join(gr2)
```

**Description**

This set of functions provide wrappers to a number of igraphs graph statistic algorithms. As for the other wrappers provided, they are intended for use inside the tidygraph framework and it is thus not necessary to supply the graph being computed on as the context is known. All of these functions are guarantied to return scalars making it easy to compute with them.

**Usage**

```r
graph_adhesion()

graph_assortativity(attr, in_attr = NULL, directed = TRUE)

graph_automorphisms(sh = "fm")
```
graph_clique_num()

graph_clique_count(min = NULL, max = NULL, subset = NULL)

graph_component_count(type = "weak")

graph_motif_count(size = 3, cut.prob = rep(0, size))

graph_diameter(weights = NULL, directed = TRUE, unconnected = TRUE)

graph_girth()

graph_radius(mode = "out")

graph Mutual_count()

graph_asym_count()

graph_unconn_count()

graph_size()

graph_order()

graph reciprocity( ignore_loops = TRUE, ratio = FALSE)

graph_min_cut(capacity = NULL)

graph_mean_dist(directed = TRUE, unconnected = TRUE)

graph_modularity(group, weights = NULL)

**Arguments**

<table>
<thead>
<tr>
<th>attr</th>
<th>The node attribute to measure on</th>
</tr>
</thead>
<tbody>
<tr>
<td>in_attr</td>
<td>An alternative node attribute to use for incoming node. If NULL the attribute given by type will be used</td>
</tr>
<tr>
<td>directed</td>
<td>Should a directed graph be treated as directed</td>
</tr>
<tr>
<td>min, max</td>
<td>The upper and lower bounds of the cliques to be considered.</td>
</tr>
<tr>
<td>subset</td>
<td>The indexes of the nodes to start the search from (logical or integer). If provided only the cliques containing these nodes will be counted.</td>
</tr>
<tr>
<td>type</td>
<td>The type of component to count, either ‘weak’ or ‘strong’. Ignored for undirected graphs.</td>
</tr>
</tbody>
</table>
size  The size of the motif, currently 3 and 4 are supported only.
cut.prob Numeric vector giving the probabilities that the search graph is cut at a certain level. Its length should be the same as the size of the motif (the size argument). By default no cuts are made.
weights Optional positive weight vector for calculating weighted distances. If the graph has a weight edge attribute, then this is used by default.
unconnected Logical, what to do if the graph is unconnected. If FALSE, the function will return a number that is one larger the largest possible diameter, which is always the number of vertices. If TRUE, the diameters of the connected components will be calculated and the largest one will be returned.
mode How should eccentricity be calculated. If "out" only outbound edges are followed. If "in" only inbound are followed. If "all" all edges are followed. Ignored for undirected graphs.
ignore_loops Logical. Should loops be ignored while calculating the reciprocity ratio Should the old "ratio" approach from igraph < v0.6 be used
capacity The capacity of the edges

group The node grouping to calculate the modularity on

Value
A scalar, the type depending on the function

Functions
- graph_adhesion: Gives the minimum edge connectivity. Wraps igraph::edge_connectivity()
- graph_assortativity: Measures the propensity of similar nodes to be connected. Wraps igraph::assortativity()
- graph_automorphisms: Calculate the number of automorphisms of the graph. Wraps igraph::automorphisms()
- graph_clique_num: Get the size of the largest clique. Wraps igraph::clique_num()
- graph_clique_count: Get the number of maximal cliques in the graph. Wraps igraph::count_max_cliques()
- graph_component_count: Count the number of unconnected components in the graph. Wraps igraph::count_components()
- graph_motif_count: Count the number of motifs in a graph. Wraps igraph::count_motifs()
- graph_diameter: Measures the length of the longest geodesic. Wraps igraph::diameter()
- graph_girth: Measures the length of the shortest circle in the graph. Wraps igraph::girth()
- graph_radius: Measures the smallest eccentricity in the graph. Wraps igraph::radius()
- graph_mutual_count: Counts the number of mutually connected nodes. Wraps igraph::dyad_census()
- graph_asym_count: Counts the number of asymmetrically connected nodes. Wraps igraph::dyad_census()
- graph_unconn_count: Counts the number of unconnected node pairs. Wraps igraph::dyad_census()
- graph_size: Counts the number of edges in the graph. Wraps igraph::gsize()
- graph_order: Counts the number of nodes in the graph. Wraps igraph::gorder()
• **graph_reciprocity**: Measures the proportion of mutual connections in the graph. Wraps `igraph::reciprocity()`

• **graph_min_cut**: Calculates the minimum number of edges to remove in order to split the graph into two clusters. Wraps `igraph::min_cut()`

• **graph_mean_dist**: Calculates the mean distance between all node pairs in the graph. Wraps `igraph::mean_distance()`

• **graph_modularity**: Calculates the modularity of the graph contingent on a provided node grouping

### Examples

```r
# Use e.g. to modify computations on nodes and edges
create_notable('meredith') %>%
  activate(nodes) %>%
  mutate(rel_neighbors = centrality_degree() / graph_order())
```

### Description

This set of functions lets the user query different aspects of the graph itself. They are all concerned with whether the graph implements certain properties and will all return a logical scalar.

### Usage

- `graph_is_simple()`
- `graph_is_directed()`
- `graph_is_bipartite()`
- `graph_is_connected()`
- `graph_is_tree()`
- `graph_is_forest()`
- `graph_is_dag()`
- `graph_is_chordal()`
- `graph_is_complete()`
- `graph_is_isomorphic_to(graph, method = "auto", ...)`
- `graph_is_subgraph_isomorphic_to(graph, method = "auto", ...)"`
Arguments

- **graph**: The graph to compare structure to
- **method**: The algorithm to use for comparison
- ... Arguments passed on to the comparison methods. See `igraph::is_isomorphic_to()` and `igraph::is_subgraph_isomorphic_to()`

Value

A logical scalar

Functions

- `graph_is_simple`: Is the graph simple (no parallel edges)
- `graph_is_directed`: Is the graph directed
- `graph_is_bipartite`: Is the graph bipartite
- `graph_is_connected`: Is the graph connected
- `graph_is_tree`: Is the graph a tree
- `graph_is_forest`: Is the graph an ensemble of multiple trees
- `graph_is_dag`: Is the graph a directed acyclic graph
- `graph_is_chordal`: Is the graph chordal
- `graph_is_complete`: Is the graph fully connected
- `graph_is_isomorphic_to`: Is the graph isomorphic to another graph. See `igraph::is_isomorphic_to()`
- `graph_is_subgraph_isomorphic_to`: Is the graph an isomorphic subgraph to another graph. See `igraph::is_subgraph_isomorphic_to()`

Examples

```r
gr <- create_tree(50, 4)
with_graph(gr, graph_is_tree())
```

Description

These functions are wrappers around the various clustering functions provided by igraph. As with the other wrappers they automatically use the graph that is being computed on, and otherwise passes on its arguments to the relevant clustering function. The return value is always a numeric vector of group memberships so that nodes or edges with the same number are part of the same group. Grouping is predominantly made on nodes and currently the only grouping of edges supported is biconnected components.
Usage

```r
group_components(type = "weak")

group_edge_betweenness(weights = NULL, directed = TRUE)

group_fast_greedy(weights = NULL)

group_infomap(weights = NULL, node_weights = NULL, trials = 10)

group_label_prop(weights = NULL, label = NULL, fixed = NULL)

group_leading_eigen(
  weights = NULL,
  steps = -1,
  label = NULL,
  options = igraph::arpack_defaults
)

group_louvain(weights = NULL)

group_optimal(weights = NULL)

group_spinglass(weights = NULL, ...)

group_walktrap(weights = NULL, steps = 4)

group_biconnected_component()
```

Arguments

type: The type of component to find. Either 'weak' or 'strong'

weights: The weight of the edges to use for the calculation. Will be evaluated in the context of the edge data.

directed: Should direction of edges be used for the calculations

node_weights: The weight of the nodes to use for the calculation. Will be evaluated in the context of the node data.

trials: Number of times partition of the network should be attempted

label: The initial groups of the nodes. Will be evaluated in the context of the node data.

fixed: A logical vector determining which nodes should keep their initial groups. Will be evaluated in the context of the node data.

steps: The number of steps in the random walks

options: Settings passed on to igraph::arpack()

...: arguments passed on to igraph::cluster_spinglass()
Value

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

Functions

- `group_components`: Group by connected components using `igraph::components()`
- `group_edge_betweenness`: Group densely connected nodes using `igraph::cluster_edge_betweenness()`
- `group_fast_greedy`: Group nodes by optimising modularity using `igraph::cluster_fast_greedy()`
- `group_infomap`: Group nodes by minimizing description length using `igraph::cluster_infomap()`
- `group_label_prop`: Group nodes by propagating labels using `igraph::cluster_label_prop()`
- `group_leading_eigen`: Group nodes based on the leading eigenvector of the modularity matrix using `igraph::cluster_leading_eigen()`
- `group_louvain`: Group nodes by multilevel optimisation of modularity using `igraph::cluster_louvain()`
- `group_optimal`: Group nodes by optimising the modularity score using `igraph::cluster_optimal()`
- `group_spinglass`: Group nodes using simulated annealing with `igraph::cluster_spinglass()`
- `group_walktrap`: Group nodes via short random walks using `igraph::cluster_walktrap()`
- `group_biconnected_component`: Group edges by their membership of the maximal biconnected components using `igraph::biconnected_components()`

Examples

```r
create_notable('tutte') %>%
  activate(nodes) %>%
  mutate(group = group_infomap())
```

<table>
<thead>
<tr>
<th>local_graph</th>
<th>Measures based on the neighborhood of each node</th>
</tr>
</thead>
</table>

Description

These functions wrap a set of functions that all measure quantities of the local neighborhood of each node. They all return a vector or list matching the node position.

Usage

```r
local_size(order = 1, mode = "all", mindist = 0)
local_members(order = 1, mode = "all", mindist = 0)
local_triangles()
local_ave_degree(weights = NULL)
local_transitivity(weights = NULL)
```
Arguments

**order**
Integer giving the order of the neighborhood.

**mode**
Character constant, it specifies how to use the direction of the edges if a directed graph is analyzed. For ‘out’ only the outgoing edges are followed, so all vertices reachable from the source vertex in at most order steps are counted. For ‘‘in’’ all vertices from which the source vertex is reachable in at most order steps are counted. ‘‘all’’ ignores the direction of the edges. This argument is ignored for undirected graphs.

**mindist**
The minimum distance to include the vertex in the result.

**weights**
Weight vector. If the graph has a weight edge attribute, then this is used by default. If this argument is given, then vertex strength (see strength) is used instead of vertex degree. But note that knn is still given in the function of the normal vertex degree. Weights are are used to calculate a weighted degree (also called strength) instead of the degree.

Value

A numeric vector or a list (for local_members) with elements corresponding to the nodes in the graph.

Functions

- **local_size**: The size of the neighborhood in a given distance from the node. (Note that the node itself is included unless mindist > 0). Wraps igraph::ego_size().
- **local_members**: The members of the neighborhood of each node in a given distance. Wraps igraph::ego().
- **local_triangles**: The number of triangles each node participate in. Wraps igraph::count_triangles().
- **local_ave_degree**: Calculates the average degree based on the neighborhood of each node. Wraps igraph::knn().
- **local_transitivity**: Calculate the transitivity of each node, that is, the propensity for the nodes neighbors to be connected. Wraps igraph::transitivity()

Examples

```r
# Get all neighbors of each graph
create_notable('chvatal') %>%
  activate(nodes) %>%
  mutate(neighborhood = local_members(mindist = 1))
```

# These are equivalent
```r
create_notable('chvatal') %>%
  activate(nodes) %>%
  mutate(n_neighbors = local_size(mindist = 1),
         degree = centrality_degree()) %>%
  as_tibble()
```
map_bfs

Apply a function to nodes in the order of a breath first search

Description

These functions allow you to map over the nodes in a graph, by first performing a breath first search on the graph and then mapping over each node in the order they are visited. The mapping function will have access to the result and search statistics for all the nodes between itself and the root in the search. To map over the nodes in the reverse direction use map_bfs_back().

Usage

map_bfs(root, mode = "out", unreachable = FALSE, .f, ...)
map_bfs_lgl(root, mode = "out", unreachable = FALSE, .f, ...)
map_bfs_chr(root, mode = "out", unreachable = FALSE, .f, ...)
map_bfs_int(root, mode = "out", unreachable = FALSE, .f, ...)
map_bfs_dbl(root, mode = "out", unreachable = FALSE, .f, ...)

Arguments

root  The node to start the search from
mode  How should edges be followed? 'out' only follows outbound edges, 'in' only follows inbound edges, and 'all' follows all edges. This parameter is ignored for undirected graphs.
unreachable  Should the search jump to an unvisited node if the search is completed without visiting all nodes.
.f  A function to map over all nodes. See Details
...  Additional parameters to pass to .f

Details

The function provided to .f will be called with the following arguments in addition to those supplied through ...:

- graph: The full tbl_graph object
- node: The index of the node currently mapped over
- rank: The rank of the node in the search
- parent: The index of the node that led to the current node
- before: The index of the node that was visited before the current node
- after: The index of the node that was visited after the current node.
• **dist**: The distance of the current node from the root
• **path**: A table containing node, rank, parent, before, after, dist, and result columns giving the values for each node leading to the current node. The result column will contain the result of the mapping of each node in a list.

Instead of spelling out all of these in the function it is possible to simply name the ones needed and use ... to catch the rest.

**Value**

map_bfs() returns a list of the same length as the number of nodes in the graph, in the order matching the node order in the graph (that is, not in the order they are called). map_bfs_*() tries to coerce its result into a vector of the classes logical (map_bfs_lgl), character (map_bfs_chr), integer (map_bfs_int), or double (map_bfs_dbl). These functions will throw an error if they are unsuccessful, so they are type safe.

**See Also**

Other node map functions: map_bfs_back(), map_dfs_back(), map_dfs()

**Examples**

# Accumulate values along a search
create_tree(40, children = 3, directed = TRUE) %>%
  mutate(value = round(runif(40)*100)) %>%
  mutate(value_acc = map_bfs_dbl(node_is_root(), .f = function(node, path, ...) {
    sum(.N$value[c(node, path$node)])
  })))

**Description**

These functions allow you to map over the nodes in a graph, by first performing a breath first search on the graph and then mapping over each node in the reverse order they are visited. The mapping function will have access to the result and search statistics for all the nodes following itself in the search. To map over the nodes in the original direction use map_bfs().

**Usage**

map_bfs_back(root, mode = "out", unreachable = FALSE, .f, ...)
map_bfs_back_lgl(root, mode = "out", unreachable = FALSE, .f, ...)
map_bfs_back_chr(root, mode = "out", unreachable = FALSE, .f, ...)
map_bfs_back_int(root, mode = "out", unreachable = FALSE, .f, ...)
map_bfs_back_dbl(root, mode = "out", unreachable = FALSE, .f, ...)
Arguments

root: The node to start the search from

mode: How should edges be followed? 'out' only follows outbound edges, 'in' only follows inbound edges, and 'all' follows all edges. This parameter is ignored for undirected graphs.

unreachable: Should the search jump to an unvisited node if the search is completed without visiting all nodes.

.f: A function to map over all nodes. See Details

...: Additional parameters to pass to .f

Details

The function provided to .f will be called with the following arguments in addition to those supplied through ...:

- graph: The full tbl_graph object
- node: The index of the node currently mapped over
- rank: The rank of the node in the search
- parent: The index of the node that led to the current node
- before: The index of the node that was visited before the current node
- after: The index of the node that was visited after the current node
- dist: The distance of the current node from the root
- path: A table containing node, rank, parent, before, after, dist, and result columns giving the values for each node reached from the current node. The result column will contain the result of the mapping of each node in a list.

Instead of spelling out all of these in the function it is possible to simply name the ones needed and use ... to catch the rest.

Value

map_bfs_back() returns a list of the same length as the number of nodes in the graph, in the order matching the node order in the graph (that is, not in the order they are called). map_bfs_back_*() tries to coerce its result into a vector of the classes logical (map_bfs_back_lgl), character (map_bfs_back_chr), integer (map_bfs_back_int), or double (map_bfs_back_dbl). These functions will throw an error if they are unsuccessful, so they are type safe.

See Also

Other node map functions: map_bfs(), map_dfs_back(), map_dfs()
Examples

# Collect values from children
create_tree(40, children = 3, directed = TRUE) %>%
  mutate(value = round(runif(40)*100)) %>%
  mutate(child_acc = map_bfs_back_dbl(node_is_root(), .f = function(node, path, ...) {
    if (nrow(path) == 0) .N$value[node]
    else {
      sum(unlist(path$result[path$parent == node]))
    }
  }))

map_dfs
Apply a function to nodes in the order of a depth first search

Description

These functions allow you to map over the nodes in a graph, by first performing a depth first search on the graph and then mapping over each node in the order they are visited. The mapping function will have access to the result and search statistics for all the nodes between itself and the root in the search. To map over the nodes in the reverse direction use map_dfs_back().

Usage

map_dfs(root, mode = "out", unreachable = FALSE, .f, ...)
map_dfs_lgl(root, mode = "out", unreachable = FALSE, .f, ...)
map_dfs_chr(root, mode = "out", unreachable = FALSE, .f, ...)
map_dfs_int(root, mode = "out", unreachable = FALSE, .f, ...)
map_dfs_dbl(root, mode = "out", unreachable = FALSE, .f, ...)

Arguments

root The node to start the search from
mode How should edges be followed? 'out' only follows outbound edges, 'in' only follows inbound edges, and 'all' follows all edges. This parameter is ignored for undirected graphs.
unreachable Should the search jump to an unvisited node if the search is completed without visiting all nodes.
.f A function to map over all nodes. See Details
... Additional parameters to pass to .f
Details

The function provided to \( .f \) will be called with the following arguments in addition to those supplied through \( \ldots \):

- **graph**: The full tbl_graph object
- **node**: The index of the node currently mapped over
- **rank**: The rank of the node in the search
- **rank_out**: The rank of the completion of the nodes subtree
- **parent**: The index of the node that led to the current node
- **dist**: The distance of the current node from the root
- **path**: A table containing node, rank, rank_out, parent, dist, and result columns giving the values for each node leading to the current node. The result column will contain the result of the mapping of each node in a list.

Instead of spelling out all of these in the function it is possible to simply name the ones needed and use \( \ldots \) to catch the rest.

Value

\( \text{map_dfs()} \) returns a list of the same length as the number of nodes in the graph, in the order matching the node order in the graph (that is, not in the order they are called). \( \text{map_dfs\_\*()} \) tries to coerce its result into a vector of the classes logical (\( \text{map_dfs\_lgl} \)), character (\( \text{map_dfs\_chr} \)), integer (\( \text{map_dfs\_int} \)), or double (\( \text{map_dfs\_dbl} \)). These functions will throw an error if they are unsuccessful, so they are type safe.

See Also

Other node map functions: \( \text{map_bfs\_back()} \), \( \text{map_bfs()} \), \( \text{map_dfs\_back()} \)

Examples

```r
# Add a random integer to the last value along a search
create_tree(40, children = 3, directed = TRUE) %>%
mutate(child_acc = map_dfs_int(node_is_root(), .f = function(node, path, \ldots) {
  last_val <- if (nrow(path) == 0) 0L else tail(unlist(path$result), 1)
  last_val + sample(1:10, 1)
})
```

---

**map_dfs\_back**

Apply a function to nodes in the reverse order of a depth first search

Description

These functions allow you to map over the nodes in a graph, by first performing a depth first search on the graph and then mapping over each node in the reverse order they are visited. The mapping function will have access to the result and search statistics for all the nodes following itself in the search. To map over the nodes in the original direction use \( \text{map_dfs()} \).
**map_dfs_back**

**Usage**

```r
map_dfs_back(root, mode = "out", unreachable = FALSE, .f, ...)
map_dfs_back_lgl(root, mode = "out", unreachable = FALSE, .f, ...)
map_dfs_back_chr(root, mode = "out", unreachable = FALSE, .f, ...)
map_dfs_back_int(root, mode = "out", unreachable = FALSE, .f, ...)
map_dfs_back_dbl(root, mode = "out", unreachable = FALSE, .f, ...)
```

**Arguments**

- `root`: The node to start the search from
- `mode`: How should edges be followed? 'out' only follows outbound edges, 'in' only follows inbound edges, and 'all' follows all edges. This parameter is ignored for undirected graphs.
- `unreachable`: Should the search jump to an unvisited node if the search is completed without visiting all nodes.
- `.f`: A function to map over all nodes. See Details
- `...`: Additional parameters to pass to `.f`

**Details**

The function provided to `.f` will be called with the following arguments in addition to those supplied through `...`:

- `graph`: The full `tbl_graph` object
- `node`: The index of the node currently mapped over
- `rank`: The rank of the node in the search
- `rank_out`: The rank of the completion of the nodes subtree
- `parent`: The index of the node that led to the current node
- `dist`: The distance of the current node from the root
- `path`: A table containing `node`, `rank`, `rank_out`, `parent`, `dist`, and `result` columns giving the values for each node reached from the current node. The `result` column will contain the result of the mapping of each node in a list.

Instead of spelling out all of these in the function it is possible to simply name the ones needed and use `...` to catch the rest.

**Value**

`map_dfs_back()` returns a list of the same length as the number of nodes in the graph, in the order matching the node order in the graph (that is, not in the order they are called). `map_dfs_back_*()` tries to coerce its result into a vector of the classes logical (`map_dfs_back_lgl`), character (`map_dfs_back_chr`), integer (`map_dfs_back_int`), or double (`map_dfs_back_dbl`). These functions will throw an error if they are unsuccessful, so they are type safe.
See Also

Other node map functions: `map_bfs_back()`, `map_bfs()`, `map_dfs()`

Examples

```r
# Collect values from the 2 closest layers of children in a dfs search
create_tree(40, children = 3, directed = TRUE) %>%
mutate(value = round(runif(40)*100)) %>%
mutate(child_acc = map_dfs_back(node_is_root(), .f = function(node, path, dist, ...) {
  if (nrow(path) == 0) .N$value[node]
  else {
    unlist(path$result[path$dist - dist <= 2])
  }
})
```

---

`map_local`  
Map a function over a graph representing the neighborhood of each node

Description

This function extracts the neighborhood of each node as a graph and maps over each of these neighborhood graphs. Conceptually it is similar to `igraph::local_scan()`, but it borrows the type safe versions available in `map_bfs()` and `map_dfs()`.

Usage

```r
map_local(order = 1, mode = "all", mindist = 0, .f, ...)
map_local_lgl(order = 1, mode = "all", mindist = 0, .f, ...)
map_local_chr(order = 1, mode = "all", mindist = 0, .f, ...)
map_local_int(order = 1, mode = "all", mindist = 0, .f, ...)
map_local_dbl(order = 1, mode = "all", mindist = 0, .f, ...)
```

Arguments

- `order`  
  Integer giving the order of the neighborhood.

- `mode`  
  Character constant, it specifies how to use the direction of the edges if a directed graph is analyzed. For `"out"` only the outgoing edges are followed, so all vertices reachable from the source vertex in at most `order` steps are counted. For `"in"` all vertices from which the source vertex is reachable in at most `order` steps are counted. `"all"` ignores the direction of the edges. This argument is ignored for undirected graphs.

- `mindist`  
  The minimum distance to include the vertex in the result.
A function to map over all nodes. See Details

Details

The function provided to .f will be called with the following arguments in addition to those supplied through . . .:

- neighborhood: The neighborhood graph of the node
- graph: The full tbl_graph object
- node: The index of the node currently mapped over

The neighborhood graph will contain an extra node attribute called .central_node, which will be TRUE for the node that the neighborhood is expanded from and FALSE for everything else.

Value

map_local() returns a list of the same length as the number of nodes in the graph, in the order matching the node order in the graph. map_local_*( ) tries to coerce its result into a vector of the classes logical (map_local_lgl), character (map_local_chr), integer (map_local_int), or double (map_local_dbl). These functions will throw an error if they are unsuccessful, so they are type safe.

Examples

# Smooth out values over a neighborhood
create_notable("meredith") %>%
  mutate(value = rpois(graph_order(), 5)) %>%
  mutate(value_smooth = map_local_dbl(order = 2, .f = function(neighborhood, ...) {
    mean(as_tibble(neighborhood, active = "nodes")$value)
  })))

morph

Create a temporary alternative representation of the graph to compute on

Description

The morph/unmorph verbs are used to create temporary representations of the graph, such as e.g. its search tree or a subgraph. A morphed graph will accept any of the standard dplyr verbs, and changed to the data is automatically propagated to the original graph when unmorphing. Tidygraph comes with a range of morphers, but is it also possible to supply your own. See Details for the requirement for custom morphers. The crystallise verb is used to extract the temporary graph representation into a tibble containing one separate graph per row and a name and graph column holding the name of each graph and the graph itself respectively. convert() is a shorthand for performing both morph and crystallise along with extracting a single tbl_graph (defaults to the first). For morphs were you know they only create a single graph, and you want to keep it, this is an easy way.
Usage

morph(.data, .f, ...)

unmorph(.data)

crystallise(.data)

crystallize(.data)

convert(.data, .f, ..., .select = 1, .clean = FALSE)

Arguments

.data A tbl_graph or a morphed_tbl_graph

.f A morphing function. See morphers for a list of provided one.

... Arguments passed on to the morpher

.select The graph to return during convert(). Either an index or the name as created
during crystallise().

.clean Should references to the node and edge indexes in the original graph be removed
when using convert

Details

It is only possible to change and add to node and edge data from a morphed state. Any filtering/removal of nodes and edges will not result in removal from the main graph. However, nodes and edges not present in the morphed state will be unaffected in the main graph when unmorphing
(if new columns were added during the morphed state they will be filled with NA).

Morphing an already morphed graph will unmorph prior to applying the new morph.

During a morphed state, the mapping back to the original graph is stored in .tidygraph_node_index and .tidygraph_edge_index columns. These are accessible but protected, meaning that any changes to them with e.g. mutate will be ignored. Furthermore, if the morph results in the merging of nodes and/or edges the original data is stored in a .data column. This is protected as well.

When supplying your own morphers the morphing function should accept a tbl_graph as its first input. The provided graph will already have nodes and edges mapped with a .tidygraph_node_index and .tidygraph_edge_index column. The return value must be a tbl_graph or a list of tbl_graphs and these must contain either a .tidygraph_node_index column or a .tidygraph_edge_index column (or both). Note that it is possible for the morph to have the edges mapped back to the original nodes and vice versa (e.g. as with to_linegraph). In that case the edge data in the morphed graph(s) will contain a .tidygraph_node_index column and or the node data a .tidygraph_edge_index column. If the morphing results in the collapse of multiple columns or edges the index columns should be converted to list columns mapping the new node/edge back to all the nodes/edges it represents. Furthermore the original node/edge data should be collapsed to a list of tibbles, with the row order matching the order in the index column element.

Value

A morphed_tbl_graph
**morphers**

Functions to generate alternate representations of graphs

**Examples**

```r
create_notable('meredith') %>%
  mutate(group = group_infomap()) %>%
  morph(to_contracted, group) %>%
  mutate(group_centrality = centrality_pagerank()) %>%
  unmorph()
```

**Description**

These functions are meant to be passed into `morph()` to create a temporary alternate representation of the input graph. They are thus not meant to be called directly. See below for detail of each morpher.

**Usage**

- `to_linegraph(graph)`
- `to_subgraph(graph, ..., subset_by = NULL)`
- `to_subcomponent(graph, node)`
- `to_split(graph, ..., split_by = NULL)`
- `to_components(graph, type = "weak")`  
- `to_complement(graph, loops = FALSE)`
- `to_local_neighborhood(graph, node, order = 1, mode = "all")`  
- `to_dominator_tree(graph, root, mode = "out")`  
- `to_minimum_spanning_tree(graph, weights = NULL)`  
- `to_shortest_path(graph, from, to, mode = "out", weights = NULL)`
- `to_bfs_tree(graph, root, mode = "out", unreachable = FALSE)`
- `to_dfs_tree(graph, root, mode = "out", unreachable = FALSE)`
- `to_simple(graph, remove_multiples = TRUE, remove_loops = TRUE)`
- `to_contracted(graph, ..., simplify = TRUE)`
- `to_unfolded_tree(graph, root, mode = "out")`
to_directed(graph)

to_undirected(graph)

to_hierarchical_clusters(graph, method = "walktrap", weights = NULL, ...)

Arguments

graph  
A tbl_graph

...  
Arguments to pass on to filter(), group_by(), or the cluster algorithm (see igraph::cluster_walktrap(), igraph::cluster_leading_eigen(), and igraph::cluster_edge_betweenness())

subset_by, split_by  
Whether to create subgraphs based on nodes or edges

node  
The center of the neighborhood for to_local_neighborhood() and the node to that should be included in the component for to_subcomponent()

type  
The type of component to split into. Either 'weak' or 'strong'

loops  
Should loops be included. Defaults to FALSE

order  
The radius of the neighborhood

mode  
How should edges be followed? 'out' only follows outbound edges, 'in' only follows inbound edges, and 'all' follows all edges. This parameter is ignored for undirected graphs.

root  
The root of the tree

weights  
Optional edge weights for the calculations

from, to  
The start and end node of the path

unreachable  
Should the search jump to a node in a new component when stuck.

remove_multiples  
Should edges that run between the same nodes be reduced to one

remove_loops  
Should edges that start and end at the same node be removed

simplify  
Should edges in the contracted graph be simplified? Defaults to TRUE

method  
The clustering method to use. Either 'walktrap', 'leading_eigen', or 'edge_betweenness'

Value

A list of tbl_graphs

Functions

- to_linegraph: Convert a graph to its line graph. When unmorphing node data will be merged back into the original edge data. Edge data will be ignored.
- to_subgraph: Convert a graph to a single subgraph. ... is evaluated in the same manner as filter. When unmorphing all data in the subgraph will get merged back.
- to_subcomponent: Convert a graph to a single component containing the specified node
• **to_split**: Convert a graph into a list of separate subgraphs. ... is evaluated in the same manner as *group_by*. When unmorphing all data in the subgraphs will get merged back, but in the case of *split_by = 'edges'* only the first instance of node data will be used (as the same node can be present in multiple subgraphs).

• **to_components**: Split a graph into its separate components. When unmorphing all data in the subgraphs will get merged back.

• **to_complement**: Convert a graph into its complement. When unmorphing only node data will get merged back.

• **to_local_neighborhood**: Convert a graph into the local neighborhood around a single node. When unmorphing all data will get merged back.

• **to_dominator_tree**: Convert a graph into its dominator tree based on a specific root. When unmorphing only node data will get merged back.

• **to_minimum_spanning_tree**: Convert a graph into its minimum spanning tree/forest. When unmorphing all data will get merged back.

• **to_shortest_path**: Limit a graph to the shortest path between two nodes. When unmorphing all data is merged back.

• **to_bfs_tree**: Convert a graph into a breath-first search tree based on a specific root. When unmorphing only node data is merged back.

• **to_dfs_tree**: Convert a graph into a depth-first search tree based on a specific root. When unmorphing only node data is merged back.

• **to_simple**: Collapse parallel edges and remove loops in a graph. When unmorphing all data will get merged back.

• **to_contracted**: Combine multiple nodes into one. ... is evaluated in the same manner as *group_by*. When unmorphing all data will get merged back.

• **to_unfolded_tree**: Unfold a graph to a tree or forest starting from multiple roots (or one), potentially duplicating nodes and edges.

• **to_directed**: Make a graph directed in the direction given by from and to

• **to_undirected**: Make a graph undirected

• **to_hierarchical_clusters**: Convert a graph into a hierarchical clustering based on a grouping

### Examples

```r
# Compute only on a subgraph of every even node
create_notable('meredith') %>%
morph(to_subgraph, seq_len(graph_order()) %% 2 == 0) %>%
mutate(neighbour_count = centrality_degree()) %>%
unmorph()
```
Querying node measures

Description

These functions are a collection of node measures that do not really fall into the class of centrality measures. For lack of a better place they are collected under the node_* umbrella of functions.

Usage

```r
node_eccentricity(mode = "out")
node_constraint(weights = NULL)
node_coreness(mode = "out")
node_diversity(weights)
node_bridging_score()
node_effective_network_size()
node_connectivity_impact()
node_closeness_impact()
node_fareness_impact()
```

Arguments

- **mode**: The way edges should be followed in the case of directed graphs.
- **weights**: The weights to use for each node during calculation

Value

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

Functions

- `node_eccentricity`: measure the maximum shortest path to all other nodes in the graph
- `node_constraint`: measures Burts constraint of the node. See `igraph::constraint()`
- `node_coreness`: measures the coreness of each node. See `igraph::coreness()`
- `node_diversity`: measures the diversity of the node. See `igraph::diversity()`
- `node_bridging_score`: measures Valente’s Bridging measures for detecting structural bridges (`influenceR`)
node_effective_network_size: measures Burt’s Effective Network Size indicating access to structural holes in the network (influenceR)

node_connectivity_impact: measures the impact on connectivity when removing the node (NetSwan)

node_closeness_impact: measures the impact on closeness when removing the node (NetSwan)

node_fareness_impact: measures the impact on fareness (distance between all node pairs) when removing the node (NetSwan)

Examples

```r
# Calculate Burt's Constraint for each node
create_notable('meredith') %>%
  mutate(b_constraint = node_constraint())
```

describe

**Calculate node ranking**

**Description**

This set of functions tries to calculate a ranking of the nodes in a graph so that nodes sharing certain topological traits are in proximity in the resulting order. These functions are of great value when composing matrix layouts and arc diagrams but could concievably be used for other things as well.

**Usage**

```r
node_rank_hclust(
  method = "average",
  dist = "shortest",
  mode = "out",
  weights = NULL,
  algorithm = "automatic",
)
```

```r
node_rank_anneal(
  cool = 0.5,
  tmin = 1e-04,
  swap_to_inversion = 0.5,
  step_multiplier = 100,
  reps = 1,
  dist = "shortest",
  mode = "out",
  weights = NULL,
  algorithm = "automatic",
)
```

```r
node_rank_branch_bound(
```
weighted_gradient = FALSE,
dist = "shortest",
mode = "out",
weights = NULL,
algorithm = "automatic"
)

node_rank_traveller(
    method = "two_opt",
    ...
)

node_rank_two(
    dist = "shortest",
    mode = "out",
    weights = NULL,
algorithm = "automatic"
)

node_rank_mds(
    method = "cmdscale",
    dist = "shortest",
    mode = "out",
    weights = NULL,
algorithm = "automatic"
)

node_rank_leafsort(
    method = "average",
    type = "OLO",
    dist = "shortest",
    mode = "out",
    weights = NULL,
algorithm = "automatic"
)

node_rank_visual(
    dist = "shortest",
    mode = "out",
    weights = NULL,
algorithm = "automatic"
)

node_rank_spectral(
node_rank

normalized = FALSE,
dist = "shortest",
mode = "out",
weights = NULL,
algorithm = "automatic"
)

node_rank_spin_out(
  step = 25,
nstart = 10,
dist = "shortest",
mode = "out",
weights = NULL,
algorithm = "automatic"
)

node_rank_spin_in(
  step = 5,
sigma = seq(20, 1, length.out = 10),
dist = "shortest",
mode = "out",
weights = NULL,
algorithm = "automatic"
)

node_rank_quadratic(
  criterion = "2SUM",
  reps = 1,
  step = 2 * graph_order(),
  step_multiplier = 1.1,
  temp_multiplier = 0.5,
  maxsteps = 50,
dist = "shortest",
mode = "out",
weights = NULL,
algorithm = "automatic"
)

node_rank_genetic(
  ..., 
dist = "shortest",
mode = "out",
weights = NULL,
algorithm = "automatic"
)

node_rank_dendser(
  ..., 
)
```r
dist = "shortest",
mode = "out",
weights = NULL,
algorithm = "automatic"
)
```

### Arguments

- **method**
  
  The method to use. See *Functions section for reference*

- **dist**
  
  The algorithm to use for deriving a distance matrix from the graph. One of
  
  - "shortest" (default): Use the shortest path between all nodes
  - "euclidean": Calculate the L2 norm on the adjacency matrix of the graph
  - "manhattan": Calculate the L1 norm on the adjacency matrix of the graph
  - "maximum": Calculate the supremum norm on the adjacency matrix of the graph
  - "canberra": Calculate a weighted manhattan distance on the adjacency matrix of the graph
  - "binary": Calculate distance as the proportion of agreement between nodes based on the adjacency matrix of the graph

  or a function that takes a tbl_graph and return a dist object with a size matching the order of the graph.

- **mode**
  
  Which edges should be included in the distance calculation. For distance measures based on the adjacency matrix, 'out' will use the matrix as is, 'in' will use the transpose, and 'all' will take the mean of the two. Defaults to 'out'. Ignored for undirected graphs.

- **weights**
  
  An edge variable to use as weight for the shortest path calculation if dist = 'shortest'

- **algorithm**
  
  The algorithm to use for the shortest path calculation if dist = 'shortest'

- **cool**
  
  Cooling rate

- **tmin**
  
  Minimum temperature

- **swap_to_inversion**
  
  Proportion of swaps in local neighborhood search

- **step_multiplier**
  
  Multiplication factor for number of iterations per temperature

- **reps**
  
  Number of repeats with random initialisation

- **weighted_gradient**
  
  Minimize the weighted gradient measure? Defaults to FALSE

- **...**
  
  Arguments passed on to other algorithms. See *Functions section for reference*

- **type**
  
  The type of leaf reordering, either 'GW' to use the "GW" method or 'OLO' to use the "OLO" method (both in seriation)

- **normalized**
  
  Should the normalized laplacian of the similarity matrix be used?

- **step**
  
  The number of iterations to run per initialisation

- **nstart**
  
  The number of random initialisations to perform
node_rank

sigma  The variance around the diagonal to use for the weight matrix. Either a single
       number or a decreasing sequence.
criterion The criterion to minimize. Either "LS" (Linear Seriation Problem), "2SUM"
       (2-Sum Problem), "BAR" (Banded Anti-Robinson form), or "Inertia" (Inertia
       criterion)
temp_multiplier Temperature multiplication factor between 0 and 1
maxsteps  The upper bound of iterations

Value
An integer vector giving the position of each node in the ranking

Functions
- node_rank_hclust: Use hierarchical clustering to rank nodes (see stats::hclust() for
  allowed methods)
- node_rank_anneal: Use simulated annealing based on the "ARSA" method in seriation
- node_rank_branch_bound: Use branch and bounds strategy to minimize the gradient mea-
  sure (only feasible for small graphs). Will use "BBURCG" or "BBWRCG" in seriation
  dependent on the weighted_gradient argument
- node_rank_traveller: Minimize hamiltonian path length using a travelling salesperson
  solver. See the the solve_TSP function in TSP for an overview of possible arguments
- node_rank_two: Use Rank-two ellipse seriation to rank the nodes. Uses "R2E" method in
  seriation
- node_rank_mds: Rank by multidimensional scaling onto one dimension. method = 'cmdscale'
  will use the classic scaling from stats, method = 'isoMDS' will use isoMDS from MASS, and
  method = 'sammon' will use sammon from MASS
- node_rank_leafsort: Minimize hamiltonian path length by reordering leafs in a hierarchical
  clustering. Method refers to the clustering algorithm (either 'average', 'single', 'complete', or
  'ward')
- node_rank_visual: Use Prim's algorithm to find a minimum spanning tree giving the rank.
  Uses the "VAT" method in seriation
- node_rank_spectral: Minimize the 2-sum problem using a relaxation approach. Uses the
  "Spectral" or "Spectral_norm" methods in seriation depending on the value of the norm
  argument
- node_rank_spin_out: Sorts points into neighborhoods by pushing large distances away from
  the diagonal. Uses the "SPIN_STS" method in seriation
- node_rank_spin_in: Sorts points into neighborhoods by concentrating low distances around
  the diagonal. Uses the "SPIN_NH" method in seriation
- node_rank_quadratic: Use quadratic assignment problem formulations to minimize cri-
  terions using simulated annealing. Uses the "QAP_LS", "QAP_2SUM", "QAP_BAR", or
  "QAP_Inertia" methods from seriation dependant on the criterion argument
- node_rank_genetic: Optimizes different criteria based on a genetic algorithm. Uses the
  "GA" method from seriation. See register_GA for an overview of relevant arguments
node_rank_dendser: Optimizes different criteria based on heuristic dendrogram seriation. Uses the "DendSer" method from seriation. See register_DendSer for an overview of relevant arguments.

Examples

```r
graph <- create_notable('zachary') %>%
  mutate(rank = node_rank_hclust())
```

node_topology

Node properties related to the graph topology

Description

These functions calculate properties that are dependent on the overall topology of the graph.

Usage

```r
node_dominator(root, mode = "out")
node_topo_order(mode = "out")
```

Arguments

- **root**: The node to start the dominator search from
- **mode**: How should edges be followed. Either 'in' or 'out'

Value

A vector of the same length as the number of nodes in the graph

Functions

- **node_dominator**: Get the immediate dominator of each node. Wraps igraph::dominator_tree().
- **node_topo_order**: Get the topological order of nodes in a DAG. Wraps igraph::topo_sort().

Examples

```r
# Sort a graph based on its topological order
create_tree(10, 2) %>%
  arrange(sample(graph_order())) %>%
  mutate(old_ind = seq_len(graph_order())) %>%
  arrange(node_topo_order())
```
**Description**

These functions all let the user query whether each node is of a certain type. All of the functions return a logical vector indicating whether the node is of the type in question. Do note that the types are not mutually exclusive and that nodes can thus be of multiple types.

**Usage**

```r
node_is_cut()
node_is_root()
node_is_leaf()
node_is_sink()
node_is_source()
node_is_isolated()
node_is_universal(mode = "out")
node_is_simplical(mode = "out")
node_is_center(mode = "out")
node_is_adjacent(to, mode = "all", include_to = TRUE)
node_is_keyplayer(k, p = 0, tol = 1e-04, maxsec = 120, roundsec = 30)
```

**Arguments**

- `mode` The way edges should be followed in the case of directed graphs.
- `to` The nodes to test for adjacency to
- `include_to` Should the nodes in `to` be marked as adjacent as well
- `k` The number of keyplayers to identify
- `p` The probability to accept a lesser state
- `tol` Optimisation tolerance, below which the optimisation will stop
- `maxsec` The total computation budget for the optimization, in seconds
- `roundsec` Number of seconds in between synchronizing workers’ answer
Value

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

Functions

- `node_is_cut`: is the node a cut node (articulation node)
- `node_is_root`: is the node a root in a tree
- `node_is_leaf`: is the node a leaf in a tree
- `node_is_sink`: does the node only have incoming edges
- `node_is_source`: does the node only have outgoing edges
- `node_is_isolated`: is the node unconnected
- `node_is_universal`: is the node connected to all other nodes in the graph
- `node_is_simplical`: are all the neighbors of the node connected
- `node_is_center`: does the node have the minimal eccentricity in the graph
- `node_is_adjacent`: is a node adjacent to any of the nodes given in to
- `node_is_keyplayer`: Is a node part of the keyplayers in the graph (influenceR)

Examples

```r
# Find the root and leaves in a tree
create_tree(40, 2) %>%
  mutate(root = node_is_root(), leaf = node_is_leaf())
```

pair_measures  
Calculate node pair properties

Description

This set of functions can be used for calculations that involve node pairs. If the calculateable measure is not symmetric the function will come in two flavours, differentiated with `_to/_from` suffix. The `_to()` functions will take the provided node indexes as the target node (recycling if necessary). For the `_from()` functions the provided nodes are taken as the source. As for the other wrappers provided, they are intended for use inside the tidygraph framework and it is thus not necessary to supply the graph being computed on as the context is known.

Usage

- `node_adhesion_to(nodes)`
- `node_adhesion_from(nodes)`
- `node_cohesion_to(nodes)`
- `node_cohesion_from(nodes)`
pair_measures

node_distance_to(nodes, mode = "out", weights = NULL, algorithm = "automatic")

node_distance_from(
    nodes,
    mode = "out",
    weights = NULL,
    algorithm = "automatic"
)

node_cocitation_with(nodes)

node_bibcoupling_with(nodes)

node_similarity_with(nodes, mode = "out", loops = FALSE, method = "jaccard")

node_max_flow_to(nodes, capacity = NULL)

node_max_flow_from(nodes, capacity = NULL)

Arguments

- **nodes**: The other part of the node pair (the first part is the node defined by the row). Recycled if necessary.
- **mode**: How should edges be followed? If 'all' all edges are considered, if 'in' only inbound edges are considered, and if 'out' only outbound edges are considered.
- **weights**: The weights to use for calculation.
- **algorithm**: The distance algorithms to use. By default it will try to select the fastest suitable algorithm. Possible values are "automatic", "unweighted", "dijkstra", "bellman-ford", and "johnson".
- **loops**: Should loop edges be considered.
- **method**: The similarity measure to calculate. Possible values are: "jaccard", "dice", and "invlogweighted".
- **capacity**: The edge capacity to use.

Value

A numeric vector of the same length as the number of nodes in the graph

Functions

- **node_adhesion_to**: Calculate the adhesion to the specified node. Wraps `igraph::edge_connectivity()`.
- **node_adhesion_from**: Calculate the adhesion from the specified node. Wraps `igraph::edge_connectivity()`.
- **node_cohesion_to**: Calculate the cohesion to the specified node. Wraps `igraph::vertex_connectivity()`.
- **node_cohesion_from**: Calculate the cohesion from the specified node. Wraps `igraph::vertex_connectivity()`.
- **node_distance_to**: Calculate various distance metrics between node pairs. Wraps `igraph::distances()`.
• node_distance_from: Calculate various distance metrics between node pairs. Wraps `igraph::distances()`
• node_cocitation_with: Calculate node pair cocitation count. Wraps `igraph::cocitation()`
• node_bibcoupling_with: Calculate node pair bibliographic coupling. Wraps `igraph::bibcoupling()`
• node_similarity_with: Calculate various node pair similarity measures. Wraps `igraph::similarity()`
• node_max_flow_to: Calculate the maximum flow to a node. Wraps `igraph::max_flow()`
• node_max_flow_from: Calculate the maximum flow from a node. Wraps `igraph::max_flow()`

Examples

```r
# Calculate the distance to the center node
create_notable('meredith') %>%
  mutate(dist_to_center = node_distance_to(node_is_center()))
```

reroute

Change terminal nodes of edges

Description

The reroute verb lets you change the beginning and end node of edges by specifying the new indexes of the start and/or end node(s). Optionally only a subset of the edges can be rerouted using the subset argument, which should be an expression that are to be evaluated in the context of the edge data and should return an index compliant vector (either logical or integer).

Usage

reroute(.data, from = NULL, to = NULL, subset = NULL)

Arguments

- `.data`: A tbl_graph or morphed_tbl_graph object. grouped_tbl_graph will be ungrouped prior to rerouting
- `from, to`: The new indexes of the terminal nodes. If NULL nothing will be changed
- `subset`: An expression evaluating to an indexing vector in the context of the edge data.

Value

An object of the same class as `.data`

Examples

```r
# Switch direction of edges
create_notable('meredith') %>%
  activate(edges) %>%
  reroute(from = to, to = from)
```

```r
# Using subset
create_notable('meredith') %>%
  activate(edges) %>%
  reroute(from = 1, subset = to > 10)
```
**sampling_games**

Graph games based on direct sampling

**Description**

This set of graph games creates graphs directly through sampling of different attributes, topologies, etc. The nature of their algorithm is described in detail at the linked igraph documentation.

**Usage**

```
play_degree(out_degree, in_degree = NULL, method = "simple")
play_dotprod(position, directed = TRUE)
play_fitness(m, out_fit, in_fit = NULL, loops = FALSE, multiple = FALSE)
play_fitness_power(
  n,
  m,
  out_exp,
  in_exp = -1,
  loops = FALSE,
  multiple = FALSE,
  correct = TRUE
)
play_erdos_renyi(n, p, m, directed = TRUE, loops = FALSE)
play_geometry(n, radius, torus = FALSE)
```

**Arguments**

- `out_degree, in_degree` The degrees of each node in the graph
- `method` The algorithm to use for the generation. Either 'simple', 'vl', or 'simple.no.multiple'
- `position` The latent position of each node by column.
- `directed` Should the resulting graph be directed.
- `m` The number of edges in the graph.
- `out_fit, in_fit` The fitness of each node.
- `loops` Are loop edges allowed.
- `multiple` Are multiple edges allowed.
- `n` The number of nodes in the graph.
- `out_exp, in_exp` Power law exponent of degree distribution.
correct Use finite size correction
p The probability of an edge occurring
radius The radius within which vertices are connected
torus Should the vertices be distributed on a torus instead of a plane

Value
A tbl_graph object

Functions

• play_degree: Create graphs based on the given node degrees. See \texttt{igraph::sample_degseq()}
• play_dotprod: Create graphs with link probability given by the dot product of the latent position of terminating nodes. See \texttt{igraph::sample_dot_product()}
• play_fitness: Create graphs where edge probabilities are proportional to terminal node fitness scores. See \texttt{igraph::sample_fitness()}
• play_fitness_power: Create graphs with an expected power-law degree distribution. See \texttt{igraph::sample_fitness_pl()}
• play_erdos_renyi: Create graphs with a fixed edge probability or count. See \texttt{igraph::sample_gnp()} and \texttt{igraph::sample_gnm()}
• play_geometry: Create graphs by positioning nodes on a plane or torus and connecting nearby ones. See \texttt{igraph::sample_grg()}

See Also
Other graph games: \texttt{component_games, evolution_games, type_games}

Examples

\texttt{plot(play_erdos_renyi(20, 0.3))}

---

**search_graph**

\textit{Search a graph with depth first and breadth first}

Description

These functions wrap the \texttt{igraph::bfs()} and \texttt{igraph::dfs()} functions to provide a consistent return value that can be used in \texttt{dplyr::mutate()} calls. Each function returns an integer vector with values matching the order of the nodes in the graph.
Usage

bfs_rank(root, mode = "out", unreachable = FALSE)
bfs_parent(root, mode = "out", unreachable = FALSE)
bfs_before(root, mode = "out", unreachable = FALSE)
bfs_after(root, mode = "out", unreachable = FALSE)
bfs_dist(root, mode = "out", unreachable = FALSE)
dfs_rank(root, mode = "out", unreachable = FALSE)
dfs_rank_out(root, mode = "out", unreachable = FALSE)
dfs_parent(root, mode = "out", unreachable = FALSE)
dfs_dist(root, mode = "out", unreachable = FALSE)

Arguments

root The node to start the search from
mode How edges are followed in the search if the graph is directed. "out" only follows outbound edges, "in" only follows inbound edges, and "all" or "total" follows all edges. This is ignored for undirected graphs.
unreachable Should the search jump to a new component if the search is terminated without all nodes being visited? Default to FALSE (only reach connected nodes).

Value
An integer vector, the nature of which is determined by the function.

Functions

- bfs_rank: Get the succession in which the nodes are visited in a breath first search
- bfs_parent: Get the nodes from which each node is visited in a breath first search
- bfs_before: Get the node that was visited before each node in a breath first search
- bfs_after: Get the node that was visited after each node in a breath first search
- bfs_dist: Get the number of nodes between the root and each node in a breath first search
- dfs_rank: Get the succession in which the nodes are visited in a depth first search
- dfs_rank_out: Get the succession in which each nodes subtree is completed in a depth first search
- dfs_parent: Get the nodes from which each node is visited in a depth first search
- dfs_dist: Get the number of nodes between the root and each node in a depth first search
Examples

# Get the depth of each node in a tree
create_tree(10, 2) %>%
  activate(nodes) %>%
  mutate(depth = bfs_dist(root = 1))

# Reorder nodes based on a depth first search from node 3
create_notable('franklin') %>%
  activate(nodes) %>%
  mutate(order = dfs_rank(root = 3)) %>%
  arrange(order)

---

**type_games**

*Graph games based on different node types*

Description

This set of games are build around different types of nodes and simulating their interaction. The nature of their algorithm is described in detail at the linked igraph documentation.

Usage

```r
play_preference(
  n,
  n_types,
  p_type = rep(1, n_types),
  p_pref = matrix(1, n_types, n_types),
  fixed = FALSE,
  directed = TRUE,
  loops = FALSE
)
```

```r
play_preference_asym(
  n,
  n_types,
  p_type = matrix(1, n_types, n_types),
  p_pref = matrix(1, n_types, n_types),
  loops = FALSE
)
```

```r
play_bipartite(n1, n2, p, m, directed = TRUE, mode = "out")
```

```r
play_traits(
  n,
  n_types,
  growth = 1,
```
\begin{verbatim}
p_type = rep(1, n_types),
p_pref = matrix(1, n_types, n_types),
callaway = TRUE,
directed = TRUE
)

play_citation_type(
  n,
  growth,
  types = rep(0, n),
  p_pref = rep(1, length(unique(types))),
  directed = TRUE
)
\end{verbatim}

**Arguments**

- `n, n1, n2` The number of nodes in the graph. For bipartite graphs `n1` and `n2` specifies the number of nodes of each type.
- `n_types` The number of different node types in the graph
- `p_type` The probability that a node will be the given type. Either a vector or a matrix, depending on the game
- `p_pref` The probability that an edge will be made to a type. Either a vector or a matrix, depending on the game
- `fixed` Should `n_types` be understood as a fixed number of nodes for each type rather than as a probability
- `directed` Should the resulting graph be directed
- `loops` Are loop edges allowed
- `p` The probability of an edge occurring
- `m` The number of edges in the graph
- `mode` The flow direction of edges
- `growth` The number of edges added at each iteration
- `callaway` Use the callaway version of the trait based game
- `types` The type of each node in the graph, enumerated from 0

**Value**

A tbl_graph object

**Functions**

- `play_preference`: Create graphs by linking nodes of different types based on a defined probability. See `igraph::sample_pref()`
- `play_preference_asym`: Create graphs by linking nodes of different types based on an asymmetric probability. See `igraph::sample_asym_pref()`
• **play_bipartite**: Create bipartite graphs of fixed size and edge count or probability. See `igraph::sample_bipartite()`.

• **play_traits**: Create graphs by evolving a graph with type based edge probabilities. See `igraph::sample_traits()` and `igraph::sample_traits_callaway()`.

• **play_citation_type**: Create citation graphs by evolving with type based linking probability. See `igraph::sample_cit_types()` and `igraph::sample_cit_cit_types()`.

**See Also**

Other graph games: `component_games`, `evolution_games`, `sampling_games`

**Examples**

```r
plot(play_bipartite(20, 30, 0.4))
```

---

**with_graph**

Evaluate a tidygraph algorithm in the context of a graph

**Description**

All tidygraph algorithms are meant to be called inside tidygraph verbs such as `mutate()`, where the graph that is currently being worked on is known and thus not needed as an argument to the function. In the off chance that you want to use an algorithm outside of the tidygraph framework you can use `with_graph()` to set the graph context temporarily while the algorithm is being evaluated.

**Usage**

```r
with_graph(graph, expr)
```

**Arguments**

- `graph` The tbl_graph to use as context
- `expr` The expression to evaluate

**Value**

The value of `expr`

**Examples**

```r
gr <- play_erdos_renyi(10, 0.3)
with_graph(gr, centrality_degree())
```
Index

.E (context_accessors), 13
.G (context_accessors), 13
.N (context_accessors), 13
%E>%(activate), 2
%N>%(activate), 2
activate, 2
activate(), 4
active (activate), 2
as_tbl_graph (as_tbl_graph.data.frame), 3
as_tbl_graph(), 19
as_tbl_graph.data.frame, 3
as_tibble(), 4
bfs_after (search_graph), 52
bfs_before (search_graph), 52
bfs_dist (search_graph), 52
bfs_parent (search_graph), 52
bfs_rank (search_graph), 52
bind_edges (bind_graphs), 6
bind_graphs, 6
bind_nodes (bind_graphs), 6
centrality, 7, 40
centrality_alpha (centrality), 7
centrality_authority (centrality), 7
centrality_betweenness (centrality), 7
centrality_betweenness_communicability (centrality), 7
centrality_betweenness_current (centrality), 7
centrality_betweenness_network (centrality), 7
centrality_betweenness_rsp_net (centrality), 7
centrality_betweenness_rsp_simple (centrality), 7
centrality_closeness (centrality), 7
centrality_closeness_generalised (centrality), 7
centrality_closeness_harmonic (centrality), 7
centrality_closeness_residual (centrality), 7
centrality_communicability (centrality), 7
centrality_communicability_even (centrality), 7
centrality_communicability_odd (centrality), 7
centrality_decay (centrality), 7
centrality_degree (centrality), 7
centrality_edge_betweenness (centrality), 7
centrality_eigen (centrality), 7
centrality_expected (centrality), 7
centrality_hub (centrality), 7
centrality_information (centrality), 7
centrality_integration (centrality), 7
centrality_katz (centrality), 7
centrality_manual (centrality), 7
centrality_pagerank (centrality), 7
centrality_power (centrality), 7
centrality_random_walk (centrality), 7
centrality_subgraph (centrality), 7
centrality_subgraph_even (centrality), 7
centrality_subgraph_odd (centrality), 7
component_games, 12, 19, 52, 56
context_accessors, 13
convert (morph), 35
create_bipartite (create_graphs), 14
create_chordal_ring (create_graphs), 14
create_citation (create_graphs), 14
create_complete (create_graphs), 14
create_de_bruijn (create_graphs), 14
create_empty (create_graphs), 14
create_graphs, 14
igraph::cluster_spinglass(), 25, 26
igraph::cluster_walktrap(), 26, 38
igraph::cocitation(), 50
igraph::components(), 26
igraph::constraint(), 40
igraph::coreness(), 40
igraph::count_components(), 22
igraph::count_max_cliques(), 22
igraph::count_motifs(), 27
igraph::count_triangles(), 27
igraph::degree(), 11
igraph::dfs(), 52
igraph::diameter(), 22
igraph::distances(), 49, 50
igraph::diversity(), 40
igraph::dominator_tree(), 46
igraph::dyad_census(), 22
igraph::edge_betweenness(), 22, 49
igraph::ego(), 27
igraph::ego_size(), 27
igraph::eigen_centrality(), 22
igraph::ego_size(), 27
igraph::estimate_betweenness(), 10
igraph::estimate_closeness(), 10
igraph::girth(), 22
igraph::gorder(), 22
igraph::gsize(), 22
igraph::hub_score(), 11
igraph::is_isomorphic_to(), 24
igraph::is_subgraph_isomorphic_to(), 24
igraph::knn(), 27
igraph::local_scan(), 34
igraph::make_chordal_ring(), 15
igraph::make_graph(), 15
igraph::max_flow(), 50
igraph::mean_distance(), 23
igraph::min_cut(), 23
igraph::page_rank(), 11
igraph::power_centrality(), 10
igraph::radius(), 22
igraph::reciprocity(), 23
igraph::sample_asym_pref(), 55
igraph::sample_bipartite(), 56
igraph::sample_cit_cit_types(), 56
igraph::sample_cit_types(), 56
igraph::sample_degseq(), 52
igraph::sample_dot_product(), 52
igraph::sample_fitness(), 52
igraph::sample_fitness_pl(), 52
igraph::sample_forestfire(), 19
igraph::sample_gnm(), 52
igraph::sample_gnp(), 52
igraph::sample_grg(), 52
igraph::sample_growing(), 19
igraph::sample_hierarchical_sbm(), 13
igraph::sample_islands(), 13
igraph::sample_last_cit(), 19
igraph::sample_pa(), 19
igraph::sample_pa_age(), 19
igraph::sample_pref(), 55
igraph::sample_sbm(), 13
igraph::sample_smallworld(), 13
igraph::sample_traits(), 56
igraph::sample_traits_callaway(), 56
igraph::sample_traits_callaway(), 56
igraph::similarity(), 50
igraph::strength(), 11
igraph::subgraph_centrality(), 27
igraph::topo_sort(), 46
igraph::transitivity(), 27
igraph::vertex_connectivity(), 49
is.tbl_graph(as_tbl_graph.data.frame), 3
local_ave_degree(local_graph), 26
local_graph, 26
local_members(local_graph), 26
local_size(local_graph), 26
local_transitivity(local_graph), 26
local_triangles(local_graph), 26
map_bfs, 28, 30, 32, 34
map_bfs(), 29, 34
map_bfs_back, 29, 29, 32, 34
map_bfs_back(), 28
map_bfs_back_chr(map_bfs_back), 29
map_bfs_back_dbl(map_bfs_back), 29
map_bfs_back_int(map_bfs_back), 29
map_bfs_back_lgl(map_bfs_back), 29
map_bfs_chr(map_bfs), 28
map_bfs_dbl(map_bfs), 28
map_bfs_int(map_bfs), 28
map_bfs_lgl(map_bfs), 28
map_dfs, 29, 30, 31, 34
map_dfs(), 32, 34
map_dfs_back, 29, 30, 32, 32
map_dfs_back(), 31
map dfs back chr (map dfs back), 32
map dfs back dbl (map dfs back), 32
map dfs back int (map dfs back), 32
map dfs back lgl (map dfs back), 32
map dfs chr (map dfs), 31
map dfs dbl (map dfs), 31
map dfs int (map dfs), 31
map dfs lgl (map dfs), 31
map local, 34
map local chr (map local), 34
map local dbl (map local), 34
map local int (map local), 34
map local lgl (map local), 34
morph, 35
morph(), 37
morphers, 35, 36, 37
node adhesion from (pair measures), 48
node adhesion to (pair measures), 48
node bibcoupling with (pair measures), 48
node bridging score (node measures), 40
node closeness impact (node measures), 40
node cocihension with (pair measures), 48
node cohesion from (pair measures), 48
node cohesion to (pair measures), 48
node connectivity impact (node measures), 40
node constraint (node measures), 40
node coreness (node measures), 40
node distance from (pair measures), 48
node distance to (pair measures), 48
node diversity (node measures), 40
node dominator (node topology), 46
node eccentricity (node measures), 40
node effective network size (node measures), 40
node fareness impact (node measures), 40
node is adjacent (node types), 47
node is center (node types), 47
node is cut (node types), 47
node is isolated (node types), 47
node is keyplayer (node types), 47
node is leaf (node types), 47
node is root (node types), 47
node is simplical (node types), 47
node is sink (node types), 47
node is source (node types), 47
node is universal (node types), 47
node max flow from (pair measures), 47
node max flow to (pair measures), 48
node measures, 40
node rank, 41
node rank anneal (node rank), 41
node rank branch bound (node rank), 41
node rank dendser (node rank), 41
node rank genetic (node rank), 41
node rank hclust (node rank), 41
node rank leafsort (node rank), 41
node rank mds (node rank), 41
node rank quadratic (node rank), 41
node rank spectral (node rank), 41
node rank spin in (node rank), 41
node rank spin out (node rank), 41
node rank traveller (node rank), 41
node rank two (node rank), 41
node rank visual (node rank), 41
node similarity with (pair measures), 48
node topo order (node topology), 46
node topology, 46
node types, 47
pair measures, 48
play barabasi albert (evolution games), 17
play barabasi albert aging (evolution games), 17
play bipartite (type games), 54
play blocks (component games), 12
play blocks hierarchy (component games), 12
play citation age (evolution games), 17
play citation type (type games), 54
play citation type(), 19
play degree (sampling games), 51
play dotprod (sampling games), 51
play erdos renyi (sampling games), 51
play fitness (sampling games), 51
play fitness power (sampling games), 51
play forestfire (evolution games), 17
play geometry (sampling games), 51
play growing (evolution games), 17
play islands (component games), 12
play preference (type games), 54
play preference asym (type games), 54
play smallworld (component games), 12
play traits (type games), 54
INDEX

play_traits(), 19
reroute, 50
sampling_games, 13, 19, 51, 56
search_graph, 52
stats::hclust(), 45
strength, 27

tbl_graph, 2
tbl_graph(as_tbl_graph.data.frame), 3
to_bfs_tree (morphers), 37
to_complement (morphers), 37
to_components (morphers), 37
to_contracted (morphers), 37
to_dfs_tree (morphers), 37
to_directed (morphers), 37
to_dominator_tree (morphers), 37
to_hierarchical_clusters (morphers), 37
to_linegraph, 36
to_linegraph (morphers), 37
to_local_neighborhood (morphers), 37
to_minimum_spanning_tree (morphers), 37
to_shortest_path (morphers), 37
to_simple (morphers), 37
to_split (morphers), 37
to_subcomponent (morphers), 37
to_subgraph (morphers), 37
to_undirected (morphers), 37
to_unfolded_tree (morphers), 37
type_games, 13, 19, 52, 54

unmorph (morph), 35

with_graph, 56