Package 'DiagrammeR'

March 1, 2018

Title Graph/Network Visualization

Version 1.0.0

Maintainer Richard Iannone <riannone@me.com>

Description Build graph/network structures using functions for stepwise addition and deletion of nodes and edges. Work with data available in tables for bulk addition of nodes, edges, and associated metadata. Use graph selections and traversals to apply changes to specific nodes or edges. A wide selection of graph algorithms allow for the analysis of graphs. Visualize the graphs and take advantage of any aesthetic properties assigned to nodes and edges.

Depends R (>= 3.2.0)

License MIT + file LICENSE

Imports dplyr (>= 0.7.4), downloader (>= 0.4), glue (>= 1.2.0), htmltools (>= 0.3.6), htmlwidgets (>= 1.0), igraph (>= 1.1.2), influenceR (>= 0.1.0), magrittr (>= 1.5), purrr (>= 0.2.4), RColorBrewer (>= 1.1-2), readr (>= 1.1.1), rlang (>= 0.2.0), rstudioapi (>= 0.7), rgexf (>= 0.15.3), scales (>= 0.5.0), stringr (>= 1.3.0), tibble (>= 1.4.2), tidyr (>= 0.8.0), viridis (>= 0.5.0), visNetwork (>= 2.0.3)

Suggests covr, DiagrammeRsvg, rsvg, knitr, testthat

URL https://github.com/rich-iannone/DiagrammeR

BugReports https://github.com/rich-iannone/DiagrammeR/issues

Encoding UTF-8

LazyData true

RoxygenNote 6.0.1

NeedsCompilation no

Author Richard Iannone [aut, cre]

Repository CRAN

Date/Publication 2018-03-01 18:01:14 UTC

R topics documented:

`add_balanced_tree` *Add a balanced tree to the graph*

Description

With a graph object of class `dgr_graph`, add a balanced tree to the graph.

Usage

```
add_balanced_tree(graph, k, h, type = NULL, label = TRUE, rel = NULL,
  node_aes = NULL, edge_aes = NULL, node_data = NULL, edge_data = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`k`	the branching factor for the tree.
`h`	the height of the tree.
`type`	an optional string that describes the entity type for the nodes to be added.
`label`	either a vector object of length `n` that provides optional labels for the new nodes, or, a boolean value where setting to `TRUE` ascribes node IDs to the label and `FALSE` yields a blank label.
`rel`	an optional string for providing a relationship label to all new edges created in the node tree.
`node_aes`	an optional list of named vectors comprising node aesthetic attributes. The helper function `node_aes()` is strongly recommended for use here as it contains arguments for each of the accepted node aesthetic attributes (e.g., `shape`, `style`, `color`, `fillcolor`).

edge_aes — an optional list of named vectors comprising edge aesthetic attributes. The helper function edge_aes() is strongly recommended for use here as it contains arguments for each of the accepted edge aesthetic attributes (e.g., shape, style, penwidth, color).

node_data — an optional list of named vectors comprising node data attributes. The helper function node_data() is strongly recommended for use here as it helps bind data specifically to the created nodes.

edge_data — an optional list of named vectors comprising edge data attributes. The helper function edge_data() is strongly recommended for use here as it helps bind data specifically to the created edges.

Value

a graph object of class dgr_graph.

Examples

```
# Create a new graph and
# add 2 different types of
# balanced trees of height
# 2 (branching twice) and
# different branching ratios
graph <-
  create_graph() %>%
  add_balanced_tree(
    k = 2,
    h = 2,
    type = "binary") %>%
  add_balanced_tree(
    k = 3,
    h = 2,
    type = "tertiary")

# Get some node information
# from this graph
graph %>%
  get_node_info() %>%
  head(5)

# Node and edge aesthetic and data
# attributes can be specified in
# the `node_aes`, `edge_aes`,
# `node_data`, and `edge_data`
# arguments
graph_w_attrs <-
  create_graph() %>%
  add_balanced_tree(
    k = 2,
    h = 2,
    label = c(
      "one", "two",
```

```
      "three", "four",
      "five", "six", "seven"),
    type = c(
      "a", "b", "b", "c",
      "c", "c", "c"),
    rel = "A",
    node_aes = node_aes(
      fillcolor = "steelblue"),
    node_data = node_data(
      value = c(
        1.6, 2.8, 3.4, 8.3,
        3.8, 5.2, 3.2)),
    edge_aes = edge_aes(
      color = "red",
      penwidth = 1.2))

# Get the first three rows of
# the graph's node data frame
graph_w_attrs %>%
  get_node_df() %>%
  head(3)

# Get the first three rows of
# the graph's edge data frame
graph_w_attrs %>%
  get_edge_df() %>%
  head(3)
```

`add_cycle` *Add a cycle of nodes to the graph*

Description

With a graph object of class `dgr_graph`, add a node cycle to the graph.

Usage

```
add_cycle(graph, n, type = NULL, label = TRUE, rel = NULL,
  node_aes = NULL, edge_aes = NULL, node_data = NULL, edge_data = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`n`	the number of nodes comprising the cycle.
`type`	an optional string that describes the entity type for the nodes to be added.
`label`	either a vector object of length `n` that provides optional labels for the new nodes, or, a boolean value where setting to `TRUE` ascribes node IDs to the label and `FALSE` yields a blank label.

rel	an optional string for providing a relationship label to all new edges created in the node cycle.
node_aes	an optional list of named vectors comprising node aesthetic attributes. The helper function node_aes() is strongly recommended for use here as it contains arguments for each of the accepted node aesthetic attributes (e.g., shape, style, color, fillcolor).
edge_aes	an optional list of named vectors comprising edge aesthetic attributes. The helper function edge_aes() is strongly recommended for use here as it contains arguments for each of the accepted edge aesthetic attributes (e.g., shape, style, penwidth, color).
node_data	an optional list of named vectors comprising node data attributes. The helper function node_data() is strongly recommended for use here as it helps bind data specifically to the created nodes.
edge_data	an optional list of named vectors comprising edge data attributes. The helper function edge_data() is strongly recommended for use here as it helps bind data specifically to the created edges.

Value

a graph object of class dgr_graph.

Examples

```
# Create a new graph and
# add a cycle of nodes to it
graph <-
  create_graph() %>%
  add_cycle(n = 6)

# Get node information
# from this graph
graph %>%
  get_node_info()

# Node and edge aesthetic and data
# attributes can be specified in
# the `node_aes`, `edge_aes`,
# `node_data`, and `edge_data`
# arguments

set.seed(23)

graph_w_attrs <-
  create_graph() %>%
  add_cycle(
    n = 3,
    label = c(
      "one", "two", "three"),
    type = c(
      "a", "a", "b"),
```

```
    rel = "A",
    node_aes = node_aes(
      fillcolor = "steelblue"),
    edge_aes = edge_aes(
      color = "red",
      penwidth = 1.2),
    node_data = node_data(
      value = c(
        1.6, 2.8, 3.4)),
    edge_data = edge_data(
      value =
        rnorm(
          n = 3,
          mean = 5.0,
          sd = 1.0)))

# Get the graph's node data frame
graph_w_attrs %>%
  get_node_df()

# Get the graph's edge data frame
graph_w_attrs %>%
  get_edge_df()
```

---

add_edge *Add an edge between nodes in a graph object*

Description

With a graph object of class dgr_graph, add an edge to nodes within the graph.

Usage

```
add_edge(graph, from, to, rel = NULL, edge_aes = NULL, edge_data = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
from	the outgoing node from which the edge is connected. There is the option to use a node label value here (and this must correspondingly also be done for the to argument) for defining node connections. Note that this is only possible if all nodes have distinct label values set and none exist as an empty string.
to	the incoming nodes to which each edge is connected. There is the option to use a node label value here (and this must correspondingly also be done for the from argument) for defining node connections. Note that this is only possible if all nodes have distinct label values set and none exist as an empty string.
rel	an optional string specifying the relationship between the connected nodes.

edge_aes — an optional list of named vectors comprising edge aesthetic attributes. The helper function edge_aes() is strongly recommended for use here as it contains arguments for each of the accepted edge aesthetic attributes (e.g., shape, style, penwidth, color).

edge_data — an optional list of named vectors comprising edge data attributes. The helper function edge_data() is strongly recommended for use here as it helps bind data specifically to the created edges.

Value

a graph object of class dgr_graph.

Examples

```
# Create a graph with 4 nodes
graph <-
  create_graph() %>%
  add_node(label = "one") %>%
  add_node(label = "two") %>%
  add_node(label = "three") %>%
  add_node(label = "four")

# Add an edge between those
# nodes and attach a
# relationship to the edge
graph <-
 add_edge(
   graph,
   from = 1,
   to = 2,
   rel = "A")

# Use the `get_edge_info()`
# function to verify that
# the edge has been created
graph %>%
  get_edge_info()

# Add another node and
# edge to the graph
graph <-
  graph %>%
  add_edge(
    from = 3,
    to = 2,
    rel = "A")

# Verify that the edge
# has been created by
# counting graph edges
graph %>%
  count_edges()
```

```
# Add edges by specifying
# node `label` values; note
# that all nodes must have
# unique `label` values to
# use this option
graph <-
  graph %>%
  add_edge(
    from = "three",
    to = "four",
    rel = "L") %>%
  add_edge(
    from = "four",
    to = "one",
    rel = "L")

# Use `get_edges()` to verify
# that the edges were added
graph %>%
  get_edges()

# Add edge aesthetic and data
# attributes during edge creation
graph_2 <-
  create_graph() %>%
  add_n_nodes(n = 2) %>%
  add_edge(
    from = 1,
    to = 2,
    rel = "M",
    edge_aes = edge_aes(
      penwidth = 1.5,
      color = "blue"),
    edge_data = edge_data(
      value = 4.3))

# Use the `get_edges()` function
# to verify that the attribute
# values were bound to the
# newly created edge
graph_2 %>%
  get_edge_df()
```

`add_edges_from_table` *Add edges and attributes to graph from a table*

Description

Add edges and their attributes to an existing graph object from data in a CSV file or a data frame.

Usage

```
add_edges_from_table(graph, table, from_col, to_col, from_to_map,
  rel_col = NULL, set_rel = NULL, drop_cols = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
table	either a path to a CSV file, or, a data frame object.
from_col	the name of the table column from which edges originate.
to_col	the name of the table column to which edges terminate.
from_to_map	a single character value for the mapping of the from and to columns in the external table (supplied as from_col and to_col, respectively) to a column in the graph's internal node data frame (ndf).
rel_col	an option to apply a column of data in the table as rel attribute values.
set_rel	an optional string to apply a rel attribute to all edges created from the table records.
drop_cols	an optional column selection statement for dropping columns from the external table before inclusion as attributes in the graph's internal edge data frame. Several columns can be dropped by name using the syntax col_1 & col_2 & .... Columns can also be dropped using a numeric column range with : (e.g., 5:8), or, by using the : between column names to specify the range (e.g., col_5_name:col_8_name).

Value

a graph object of class dgr_graph.

Examples

```
# Create an empty graph and then
# add nodes to it from the
# `currencies` dataset available
# in the package
graph <-
  create_graph() %>%
  add_nodes_from_table(
    table = currencies)

# Now we want to add edges to the
# graph using an included dataset,
# `usd_exchange_rates`, which has
# exchange rates between USD and
# many other currencies; the key
# here is that the data in the
# `from` and `to` columns in the
# external table maps to graph
# node data available in the
# `iso_4217_code` column of the
# graph's internal node data frame
```

```
graph_1 <-
  graph %>%
    add_edges_from_table(
      table = usd_exchange_rates,
      from_col = from_currency,
      to_col = to_currency,
      from_to_map = iso_4217_code)

# View part of the graph's
# internal edge data frame
graph_1 %>%
  get_edge_df() %>%
  head()

# If you would like to assign
# any of the table's columns as the
# `rel` attribute, this can done
# with the `rel_col` argument; to
# set a static `rel` attribute for
# all edges created, use `set_rel`
graph_2 <-
  graph %>%
    add_edges_from_table(
      table = usd_exchange_rates,
      from_col = from_currency,
      to_col = to_currency,
      from_to_map = iso_4217_code,
      set_rel = "from_usd")

# View part of the graph's internal
# edge data frame (edf)
graph_2 %>%
  get_edge_df() %>%
  head()
```

`add_edges_w_string` *Add one or more edges using a text string*

Description

With a graph object of class `dgr_graph`, add one or more edges to the graph using a text string.

Usage

```
add_edges_w_string(graph, edges, rel = NULL, use_labels = FALSE)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

edges	a single-length vector with a character string specifying the edges. For a directed graph, the string object should be formatted as a series of node ID values as `[node_ID_1]->[node_ID_2]` separated by a one or more space characters. For undirected graphs, `--` should replace `->`. Line breaks in the vector won't cause an error.
rel	an optional vector specifying the relationship between the connected nodes.
use_labels	an option to use node `label` values in the `edges` string to define node connections. Note that this is only possible if all nodes have distinct `label` values set and none exist as an empty string.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a graph with 4 nodes
graph <-
  create_graph() %>%
  add_node(label = "one") %>%
  add_node(label = "two") %>%
  add_node(label = "three") %>%
  add_node(label = "four")

# Add edges between nodes using
# a character string with node
# ID values
graph_node_id <-
  graph %>%
  add_edges_w_string(
    edges = "1->2 1->3 2->4 2->3")

# Show the graph's internal
# edge data frame
graph_node_id %>%
  get_edge_df()

# Add edges between nodes using
# a character string with node
# label values and setting
# `use_labels = TRUE`; note that
# all nodes must have unique
# `label` values to use this
graph_node_label <-
  graph %>%
  add_edges_w_string(
    edges =
      "one->two one->three
       two->four two->three",
    use_labels = TRUE)
```

```
# Show the graph's internal
# edge data frame (it's the
# same as before)
graph_node_label %>%
  get_edge_df()
```

---

add_edge_clone *Add a clone of an existing edge to the graph*

Description

Add a new edge to a graph object of class dgr_graph which is a clone of an edge already in the graph. All edge attributes are preserved.

Usage

```
add_edge_clone(graph, edge, from, to)
```

Arguments

graph	a graph object of class dgr_graph.
edge	an edge ID corresponding to the graph edge to be cloned.
from	the outgoing node from which the edge is connected.
to	the incoming nodes to which each edge is connected.

Value

a graph object of class dgr_graph.

Examples

```
# Create a graph with a path of
# 2 nodes; supply a common `rel`
# edge attribute for all edges
# in this path and then add a
# `color` edge attribute
graph <-
  create_graph() %>%
  add_path(
    n = 2,
    rel = "a") %>%
  select_last_edges_created() %>%
  set_edge_attrs(
    edge_attr = color,
    values = "steelblue") %>%
  clear_selection()

# Display the graph's internal
```

```
# edge data frame
graph %>%
  get_edge_df()

# Create a new node (will have
# node ID of `3`) and then
# create an edge between it and
# node `1` while reusing the edge
# attributes of edge `1` -> `2`
# (edge ID `1`)
graph_2 <-
  graph %>%
  add_node() %>%
  add_edge_clone(
    edge = 1,
    from = 3,
      to = 1)

# Display the graph's internal
# edge data frame
graph_2 %>%
  get_edge_df()

# The same change can be performed
# with some helper functions in the
# `add_edge_clone()` function call
graph_3 <-
  graph %>%
    add_node() %>%
    add_edge_clone(
      edge = get_last_edges_created(.),
      from = get_last_nodes_created(.),
      to = 1)

# Display the graph's internal
# edge data frame
graph_3 %>%
  get_edge_df()
```

`add_edge_df` *Add edges from an edge data frame to an existing graph object*

Description

With a graph object of class `dgr_graph`, add edges from an edge data frame to that graph.

Usage

```
add_edge_df(graph, edge_df)
```

Arguments

graph	a graph object of class dgr_graph.
edge_df	an edge data frame that is created using create_edge_df.

Value

a graph object of class dgr_graph.

Examples

```
# Create a graph with 4 nodes
# and no edges
graph <-
  create_graph() %>%
  add_n_nodes(n = 4)

# Create an edge data frame (edf)
edf <-
  create_edge_df(
    from = c(1, 2, 3),
      to = c(4, 3, 1))

# Add the edge data frame to
# the graph object to create
# a graph with both nodes
# and edges
graph <-
  graph %>%
  add_edge_df(
    edge_df = edf)

# Get the graph's edges to
# verify that the edf had
# been added
graph %>%
  get_edges(
    return_type = "vector")
```

---

add_forward_edges_ws *Add new edges with identical definitions as with a selection of edges*

Description

Add edges in the same direction of one or more edges available as an edge selection in a graph object of class dgr_graph. New graph edges have the same edge definitions as those in the selection except with new edge ID values. There is also the option to assign a common rel grouping to the newly created edges. Upon addition of the edges, the edge selection will be retained for further selection or traversal operations.

Selections of edges can be performed using the following `select_...` functions: `select_edges()`, `select_last_edge()`, or `select_edges_by_node_id()`. Selections of edges can also be performed using the following traversal functions: `trav_out_edge()`, `trav_in_edge()`, or `trav_both_edge()`.

Usage

```
add_forward_edges_ws(graph, rel = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`rel`	an optional string to apply a `rel` attribute to all newly created edges.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create an empty graph, add 2 nodes to it,
# and create the edge `1->2`
graph <-
  create_graph() %>%
  add_n_nodes(
    n = 2,
    type = "type_a",
    label = c("a_1", "a_2")) %>%
  add_edge(
    from = 1, to = 2, rel = "a")

# Get the graph's edges
graph %>%
  get_edge_ids()

# Select the edge and create 2 additional edges
# with the same definition (`1->2`) but with
# different `rel` values (`b` and `c`)
graph <-
  graph %>%
  select_edges() %>%
  add_forward_edges_ws(rel = "b") %>%
  add_forward_edges_ws(rel = "c") %>%
  clear_selection()

# Get the graph's edge data frame
graph %>%
  get_edge_df()
```

add_full_graph *Add a fully connected graph*

Description

With a graph object of class dgr_graph, add a fully connected graph either with or without loops. If the graph object set as directed, the added graph will have edges to and from each pair of nodes. In the undirected case, a single edge will link each pair of nodes.

Usage

```
add_full_graph(graph, n, type = NULL, label = TRUE, rel = NULL,
  edge_wt_matrix = NULL, keep_loops = FALSE, node_aes = NULL,
  edge_aes = NULL, node_data = NULL, edge_data = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
n	the number of nodes comprising the fully connected graph.
type	an optional string that describes the entity type for the nodes to be added.
label	either a vector object of length n that provides optional labels for the new nodes, or, a boolean value where setting to TRUE ascribes node IDs to the label and FALSE or NULL yields a blank label.
rel	an optional string for providing a relationship label to all new edges created in the connected graph.
edge_wt_matrix	an optional matrix of n by n dimensions containing values to apply as edge weights. If the matrix has row names or column names and label = TRUE, those row or column names will be used as node label values.
keep_loops	an option to simplify the fully connected graph by removing loops (edges from and to the same node). The default value is FALSE.
node_aes	an optional list of named vectors comprising node aesthetic attributes. The helper function node_aes() is strongly recommended for use here as it contains arguments for each of the accepted node aesthetic attributes (e.g., shape, style, color, fillcolor).
edge_aes	an optional list of named vectors comprising edge aesthetic attributes. The helper function edge_aes() is strongly recommended for use here as it contains arguments for each of the accepted edge aesthetic attributes (e.g., shape, style, penwidth, color).
node_data	an optional list of named vectors comprising node data attributes. The helper function node_data() is strongly recommended for use here as it helps bind data specifically to the created nodes.
edge_data	an optional list of named vectors comprising edge data attributes. The helper function edge_data() is strongly recommended for use here as it helps bind data specifically to the created edges.

Value

a graph object of class dgr_graph.

Examples

```
# Create a new graph object
# and add a directed and fully
# connected graph with 3 nodes
# and edges to and from all
# pairs of nodes; with the option
# `keep_loops = TRUE` nodes
# will also have edges from
# and to themselves
graph <-
  create_graph() %>%
  add_full_graph(
    n = 3, keep_loops = TRUE)

# Get node information
# from this graph
graph %>%
  get_node_info()

# Using `keep_loops = FALSE`
# (the default) will remove
# the loops
create_graph() %>%
  add_full_graph(n = 3) %>%
  get_node_info()

# Values can be set for
# the node `label`, node
# `type`, and edge `rel`
graph <-
  create_graph() %>%
  add_full_graph(
    n = 3,
    type = "connected",
    label = c("1st", "2nd", "3rd"),
    rel = "connected_to")

# Show the graph's node
# data frame (ndf)
graph %>%
  get_node_df()

# Show the graph's edge
# data frame (edf)
graph %>%
  get_edge_df()

# Create a fully-connected and
```

```
# directed graph with 3 nodes,
# and, where a matrix provides
# edge weights; first, create the
# matrix (with row names to be
# used as node labels)
set.seed(23)

edge_wt_matrix <-
  rnorm(100, 5, 2) %>%
  sample(9, FALSE) %>%
  round(2) %>%
  matrix(
    nc = 3,
    nr = 3,
    dimnames = list(c("a", "b", "c")))

# Create the fully-connected
# graph (without loops however)
graph <-
  create_graph() %>%
  add_full_graph(
    n = 3,
    type = "weighted",
    label = TRUE,
    rel = "related_to",
    edge_wt_matrix = edge_wt_matrix,
    keep_loops = FALSE)

# Show the graph's node
# data frame (ndf)
graph %>%
  get_node_df()

# Show the graph's edge
# data frame (edf)
graph %>%
  get_edge_df()

# An undirected graph can
# also use a matrix with
# edge weights, but only
# the lower triangle of
# that matrix will be used
create_graph(directed = FALSE) %>%
  add_full_graph(
    n = 3,
    type = "weighted",
    label = TRUE,
    rel = "related_to",
    edge_wt_matrix = edge_wt_matrix,
    keep_loops = FALSE) %>%
  get_edge_df()
```

add_global_graph_attrs *Add one or more global graph attributes*

Description

Add global attributes of a specific type (either graph_attrs, node_attrs, or edge_attrs for a graph object of class dgr_graph).

Usage

```
add_global_graph_attrs(graph, attr, value, attr_type)
```

Arguments

graph	a graph object of class dgr_graph.
attr	the name of the attribute to set for the type of global attribute specified.
value	the value to be set for the chosen attribute specified in the attr_for_type argument.
attr_type	the specific type of global graph attribute to set. The type is specified with graph, node, or edge.

Value

a graph object of class dgr_graph.

Examples

```
# Create a new graph with no
# global graph attributes and
# add a global graph attribute
graph <-
  create_graph(
    attr_theme = NULL) %>%
  add_global_graph_attrs(
    attr = "overlap",
    value = "true",
    attr_type = "graph")

# Verify that the attribute
# addition has been made
graph %>%
  get_global_graph_attr_info()

# Add another attribute with
# `add_global_graph_attrs()`
graph <-
  graph %>%
```

```
  add_global_graph_attrs(
    attr = "penwidth",
    value = 12,
    attr_type = "node")

# Verify that the attribute
# addition has been made
graph %>%
  get_global_graph_attr_info()

# When adding an attribute where
# `attr` and `attr_type` already
# exists, the value provided will
# serve as an update
graph %>%
  add_global_graph_attrs(
    attr = "penwidth",
    value = 15,
    attr_type = "node") %>%
  get_global_graph_attr_info()
```

---

add_gnm_graph *Add a G(n, m) Erdos-Renyi graph*

Description

To an existing graph object, add a graph built according to the Erdos-Renyi G(n, m) model. This uses the same constant probability when creating the fixed number of edges. Thus for `n` nodes there will be `m` edges and, if the `loops` argument is set as `TRUE`, then random loop edges will be part of `m`.

Usage

```
add_gnm_graph(graph, n, m, loops = FALSE, type = NULL, label = TRUE,
  rel = NULL, node_aes = NULL, edge_aes = NULL, node_data = NULL,
  edge_data = NULL, set_seed = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`n`	the number of nodes comprising the generated graph.
`m`	the number of edges in the generated graph.
`loops`	a logical value (default is `FALSE`) that governs whether loops are allowed to be created.
`type`	an optional string that describes the entity type for all the nodes to be added.
`label`	a boolean value where setting to `TRUE` ascribes node IDs to the label and `FALSE` yields a blank label.
`rel`	an optional string for providing a relationship label to all edges to be added.

node_aes	an optional list of named vectors comprising node aesthetic attributes. The helper function node_aes() is strongly recommended for use here as it contains arguments for each of the accepted node aesthetic attributes (e.g., shape, style, color, fillcolor).
edge_aes	an optional list of named vectors comprising edge aesthetic attributes. The helper function edge_aes() is strongly recommended for use here as it contains arguments for each of the accepted edge aesthetic attributes (e.g., shape, style, penwidth, color).
node_data	an optional list of named vectors comprising node data attributes. The helper function node_data() is strongly recommended for use here as it helps bind data specifically to the created nodes.
edge_data	an optional list of named vectors comprising edge data attributes. The helper function edge_data() is strongly recommended for use here as it helps bind data specifically to the created edges.
set_seed	supplying a value sets a random seed of the Mersenne-Twister implementation.

Examples

```
# Create an undirected GNM
# graph with 100 nodes and
# 120 edges
gnm_graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 100,
    m = 120)

# Get a count of nodes
gnm_graph %>%
  count_nodes()

# Get a count of edges
gnm_graph %>%
  count_edges()
```

---

add_gnp_graph *Add a G(n, p) Erdos-Renyi graph*

Description

To an existing graph object, add a graph built according to the Erdos-Renyi G(n, p) model, which uses a constant probability when creating edges.

Usage

```
add_gnp_graph(graph, n, p, loops = FALSE, type = NULL, label = TRUE,
  rel = NULL, node_aes = NULL, edge_aes = NULL, node_data = NULL,
  edge_data = NULL, set_seed = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
n	the number of nodes comprising the generated graph.
p	the probability of creating an edge between two arbitrary nodes.
loops	a logical value (default is FALSE) that governs whether loops are allowed to be created.
type	an optional string that describes the entity type for all the nodes to be added.
label	a boolean value where setting to TRUE ascribes node IDs to the label and FALSE yields a blank label.
rel	an optional string for providing a relationship label to all edges to be added.
node_aes	an optional list of named vectors comprising node aesthetic attributes. The helper function node_aes() is strongly recommended for use here as it contains arguments for each of the accepted node aesthetic attributes (e.g., shape, style, color, fillcolor).
edge_aes	an optional list of named vectors comprising edge aesthetic attributes. The helper function edge_aes() is strongly recommended for use here as it contains arguments for each of the accepted edge aesthetic attributes (e.g., shape, style, penwidth, color).
node_data	an optional list of named vectors comprising node data attributes. The helper function node_data() is strongly recommended for use here as it helps bind data specifically to the created nodes.
edge_data	an optional list of named vectors comprising edge data attributes. The helper function edge_data() is strongly recommended for use here as it helps bind data specifically to the created edges.
set_seed	supplying a value sets a random seed of the Mersenne-Twister implementation.

Examples

```
# Create an undirected GNP
# graph with 100 nodes using
# a probability value of 0.05
gnp_graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnp_graph(
    n = 100,
    p = 0.05)

# Get a count of nodes
gnp_graph %>%
  count_nodes()

# Get a count of edges
gnp_graph %>%
  count_edges()
```

add_graph_action *Add a graph action for execution at every transform*

Description

Add a graph function along with its arguments to be run at every graph transformation step.

Usage

```
add_graph_action(graph, fcn, ..., action_name = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
fcn	the name of the function to use.
...	arguments and values to pass to the named function in fcn, if necessary.
action_name	an optional name for labeling the action.

Value

a graph object of class dgr_graph.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 10,
    m = 22,
    set_seed = 23)

# Add a graph action that sets a node
# attr column with a function; the
# main function `set_node_attr_w_fcn()`
# uses the `get_betweenness()` function
# to provide betweenness values in the
# `btwns` column; this action will
# occur whenever there is a function
# called on the graph that modifies it
# (e.g., `add_n_nodes()`)
graph <-
  graph %>%
  add_graph_action(
    fcn = "set_node_attr_w_fcn",
    node_attr_fcn = "get_betweenness",
    column_name = "btwns",
```

```
    action_name = "get_btwns")

# To ensure that the action is
# available in the graph, use the
# `get_graph_actions()` function
graph %>%
  get_graph_actions()
```

`add_graph_to_graph_series` *Add graph object to a graph series object*

Description

Add a graph object to an extant graph series object for storage of multiple graphs across a sequential or temporal one-dimensional array.

Usage

```
add_graph_to_graph_series(graph_series, graph)
```

Arguments

`graph_series`	a graph series object to which the graph object will be added.
`graph`	a graph object to add to the graph series object.

Value

a graph series object of type `dgr_graph_1D`.

Examples

```
# Create three graphs
graph_1 <-
  create_graph() %>%
  add_path(n = 4)

graph_2 <-
  create_graph() %>%
  add_cycle(n = 5)

graph_3 <-
  create_graph() %>%
  add_star(n = 6)

# Create an empty graph series
# and add the graphs
series <-
  create_graph_series() %>%
```

```
  add_graph_to_graph_series(
    graph = graph_1) %>%
  add_graph_to_graph_series(
    graph = graph_2) %>%
  add_graph_to_graph_series(
    graph = graph_3)

# Count the number of graphs
# in the graph series
series %>%
  count_graphs_in_graph_series()
```

`add_grid_2d` *Add a 2D grid of nodes to the graph*

Description

With a graph object of class `dgr_graph`, add a two-dimensional grid to the graph.

Usage

```
add_grid_2d(graph, x, y, type = NULL, label = TRUE, rel = NULL,
  node_aes = NULL, edge_aes = NULL, node_data = NULL, edge_data = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`x`	the number of nodes in the x direction.
`y`	the number of nodes in the y direction.
`type`	an optional string that describes the entity type for the nodes to be added.
`label`	either a vector object of length `x * y` that provides optional labels for the new nodes, or, a boolean value where setting to `TRUE` ascribes node IDs to the label and `FALSE` yields a blank label.
`rel`	an optional string for providing a relationship label to all new edges created in the grid.
`node_aes`	an optional list of named vectors comprising node aesthetic attributes. The helper function `node_aes()` is strongly recommended for use here as it contains arguments for each of the accepted node aesthetic attributes (e.g., `shape`, `style`, `color`, `fillcolor`).
`edge_aes`	an optional list of named vectors comprising edge aesthetic attributes. The helper function `edge_aes()` is strongly recommended for use here as it contains arguments for each of the accepted edge aesthetic attributes (e.g., `shape`, `style`, `penwidth`, `color`).
`node_data`	an optional list of named vectors comprising node data attributes. The helper function `node_data()` is strongly recommended for use here as it helps bind data specifically to the created nodes.

edge_data — an optional list of named vectors comprising edge data attributes. The helper function edge_data() is strongly recommended for use here as it helps bind data specifically to the created edges.

Value

a graph object of class dgr_graph.

Examples

```
# Create a new graph and add
# a 3 x 3 grid
graph <-
  create_graph() %>%
  add_grid_2d(
    x = 3, y = 3,
    type = "grid")

# Get node information
# from this graph
graph %>%
  get_node_info()

# Attributes can be specified
# in extra arguments and these
# are applied in order; Usually
# these attributes are applied
# to nodes (e.g., `type` is a
# node attribute) but the `rel`
# attribute will apply to the
# edges
graph_w_attrs <-
  create_graph() %>%
  add_grid_2d(
    x = 3, y = 2,
    label = c("one", "two",
              "three", "four",
              "five", "six"),
    type = c("a", "a",
             "b", "b",
             "c", "c"),
    rel = "grid",
    node_data = node_data(
      value = c(
        1.2, 8.4, 3.4,
        5.2, 6.1, 2.6)))

# Get the graph's node data frame
graph_w_attrs %>%
  get_node_df()

# Get the graph's edge data frame
```

```
graph_w_attrs %>%
  get_edge_df()
```

`add_grid_3d` *Add a 3D grid of nodes to the graph*

Description

With a graph object of class `dgr_graph`, add a three-dimensional grid to the graph.

Usage

```
add_grid_3d(graph, x, y, z, type = NULL, label = TRUE, rel = NULL,
  node_aes = NULL, edge_aes = NULL, node_data = NULL, edge_data = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`x`	the number of nodes in the x direction.
`y`	the number of nodes in the y direction.
`z`	the number of nodes in the z direction.
`type`	an optional string that describes the entity type for the nodes to be added.
`label`	either a vector object of length `x * y * z` that provides optional labels for the new nodes, or, a boolean value where setting to `TRUE` ascribes node IDs to the label and `FALSE` yields a blank label.
`rel`	an optional string for providing a relationship label to all new edges created in the grid.
`node_aes`	an optional list of named vectors comprising node aesthetic attributes. The helper function `node_aes()` is strongly recommended for use here as it contains arguments for each of the accepted node aesthetic attributes (e.g., `shape`, `style`, `color`, `fillcolor`).
`edge_aes`	an optional list of named vectors comprising edge aesthetic attributes. The helper function `edge_aes()` is strongly recommended for use here as it contains arguments for each of the accepted edge aesthetic attributes (e.g., `shape`, `style`, `penwidth`, `color`).
`node_data`	an optional list of named vectors comprising node data attributes. The helper function `node_data()` is strongly recommended for use here as it helps bind data specifically to the created nodes.
`edge_data`	an optional list of named vectors comprising edge data attributes. The helper function `edge_data()` is strongly recommended for use here as it helps bind data specifically to the created edges.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a new graph and add
# a 2 x 2 x 2 grid
graph <-
  create_graph() %>%
  add_grid_3d(
    x = 2, y = 2, z = 2,
    type = "grid")

# Get node information
# from this graph
graph %>%
  get_node_info()

# Attributes can be specified
# in extra arguments and these
# are applied in order; Usually
# these attributes are applied
# to nodes (e.g., `type` is a
# node attribute) but the `rel`
# attribute will apply to the
# edges
graph_w_attrs <-
  create_graph() %>%
  add_grid_3d(
    x = 2, y = 2, z = 2,
    label = c(
      "one", "two", "three",
      "four", "five", "six",
      "seven", "eight"),
    type = c(
      "a", "a", "b",
      "b", "c", "c",
      "d", "d"),
    rel = "grid",
    node_data = node_data(
      value = c(
        1.2, 8.4, 3.4,
        5.2, 6.1, 2.6,
        6.3, 9.3)))

# Get the graph's node data frame
graph_w_attrs %>%
  get_node_df()

# Get the graph's edge data frame
graph_w_attrs %>%
  get_edge_df()
```

---

add_growing_graph *Create a random growing graph with m edges added per step*

Description

To an existing graph object, add a graph built by adding `m` new edges at each time step (where a node is added).

Usage

```
add_growing_graph(graph, n, m = 1, citation = FALSE, type = NULL,
  label = TRUE, rel = NULL, node_aes = NULL, edge_aes = NULL,
  node_data = NULL, edge_data = NULL, set_seed = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`n`	the number of nodes comprising the generated graph.
`m`	the number of edges added per time step.
`citation`	a logical value (default is `FALSE`) that governs whether a citation graph is to be created. This is where new edges specifically originate from the newly added node in the most recent time step.
`type`	an optional string that describes the entity type for all the nodes to be added.
`label`	a boolean value where setting to `TRUE` ascribes node IDs to the label and `FALSE` yields a blank label.
`rel`	an optional string for providing a relationship label to all edges to be added.
`node_aes`	an optional list of named vectors comprising node aesthetic attributes. The helper function `node_aes()` is strongly recommended for use here as it contains arguments for each of the accepted node aesthetic attributes (e.g., `shape`, `style`, `color`, `fillcolor`).
`edge_aes`	an optional list of named vectors comprising edge aesthetic attributes. The helper function `edge_aes()` is strongly recommended for use here as it contains arguments for each of the accepted edge aesthetic attributes (e.g., `shape`, `style`, `penwidth`, `color`).
`node_data`	an optional list of named vectors comprising node data attributes. The helper function `node_data()` is strongly recommended for use here as it helps bind data specifically to the created nodes.
`edge_data`	an optional list of named vectors comprising edge data attributes. The helper function `edge_data()` is strongly recommended for use here as it helps bind data specifically to the created edges.
`set_seed`	supplying a value sets a random seed of the `Mersenne-Twister` implementation.

Examples

```
# Create a random, growing
# citation graph with 100
# nodes, adding an edge after
# each node addition
growing_graph <-
  create_graph() %>%
```

```
  add_growing_graph(
    n = 100,
    m = 1,
    citation = TRUE,
    set_seed = 23)

# Get a count of nodes
growing_graph %>%
  count_nodes()

# Get a count of edges
growing_graph %>%
  count_edges()
```

`add_islands_graph` *Create a random islands graph with edges between the islands*

Description

To an existing graph object, add several Erdos-Renyi random graphs (the islands) using a common set of parameters, connected together by a fixed number of edges.

Usage

```
add_islands_graph(graph, n_islands, island_size, p, edges_between,
  type = NULL, label = TRUE, rel = NULL, node_aes = NULL,
  edge_aes = NULL, node_data = NULL, edge_data = NULL, set_seed = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`n_islands`	the number of islands in the generated graph.
`island_size`	the size of the islands in the generated graph.
`p`	the probability of there being edges between the islands.
`edges_between`	The number of edges between islands.
`type`	an optional string that describes the entity type for all the nodes to be added.
`label`	a boolean value where setting to `TRUE` ascribes node IDs to the label and `FALSE` yields a blank label.
`rel`	an optional string for providing a relationship label to all edges to be added.
`node_aes`	an optional list of named vectors comprising node aesthetic attributes. The helper function `node_aes()` is strongly recommended for use here as it contains arguments for each of the accepted node aesthetic attributes (e.g., `shape`, `style`, `color`, `fillcolor`).
`edge_aes`	an optional list of named vectors comprising edge aesthetic attributes. The helper function `edge_aes()` is strongly recommended for use here as it contains arguments for each of the accepted edge aesthetic attributes (e.g., `shape`, `style`, `penwidth`, `color`).

node_data	an optional list of named vectors comprising node data attributes. The helper function node_data() is strongly recommended for use here as it helps bind data specifically to the created nodes.
edge_data	an optional list of named vectors comprising edge data attributes. The helper function edge_data() is strongly recommended for use here as it helps bind data specifically to the created edges.
set_seed	supplying a value sets a random seed of the Mersenne-Twister implementation.

Examples

```
# Create a graph of islands
islands_graph <-
  create_graph() %>%
  add_islands_graph(
    n_islands = 4,
    island_size = 10,
    p = 0.5,
    edges_between = 1,
    set_seed = 23)

# Get a count of nodes
islands_graph %>%
  count_nodes()

# Get a count of edges
islands_graph %>%
  count_edges()
```

---

add_mathjax *Add MathJax-formatted equation text*

Description

Add MathJax-formatted equation text

Usage

```
add_mathjax(gv = NULL, include_mathjax = TRUE)
```

Arguments

gv	a grViz htmlwidget.
include_mathjax	logical to add mathjax JS. Change to FALSE if using with RMarkdown since MathJax will likely already be added.

Value

a grViz htmlwidget

add_node *Add a node to an existing graph object*

Description

With a graph object of class dgr_graph, add a new node to the graph. One can optionally provide node attributes for the created node. There is also the option to create edges to and from existing nodes in the graph. Because new edges can also be created through this function, there is the possibility to set edge attributes for any new graph edges.

Usage

```
add_node(graph, type = NULL, label = NULL, from = NULL, to = NULL,
  node_aes = NULL, edge_aes = NULL, node_data = NULL, edge_data = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
type	an optional character object that acts as a group identifier for the node to be added.
label	an optional character object that describes the node.
from	an optional vector containing node IDs from which edges will be directed to the new node.
to	an optional vector containing node IDs to which edges will be directed from the new node.
node_aes	an optional list of named vectors comprising node aesthetic attributes. The helper function node_aes() is strongly recommended for use here as it contains arguments for each of the accepted node aesthetic attributes (e.g., shape, style, color, fillcolor).
edge_aes	an optional list of named vectors comprising edge aesthetic attributes. The helper function edge_aes() is strongly recommended for use here as it contains arguments for each of the accepted edge aesthetic attributes (e.g., shape, style, penwidth, color).
node_data	an optional list of named vectors comprising node data attributes. The helper function node_data() is strongly recommended for use here as it helps bind data specifically to the created nodes.
edge_data	an optional list of named vectors comprising edge data attributes. The helper function edge_data() is strongly recommended for use here as it helps bind data specifically to the created edges.

Value

a graph object of class dgr_graph.

Examples

```
# Create an empty graph and add 2 nodes by using
# the `add_node()` function twice
graph <-
  create_graph() %>%
  add_node() %>%
  add_node()

# Get a count of all nodes
# in the graph
graph %>%
  count_nodes()

# The nodes added were given
# ID values `1` and `2`; obtain
# the graph's node IDs
graph %>%
  get_node_ids()

# Add a node with a `type`
# value defined
graph <-
  graph %>%
  add_node(type = "person")

# View the graph's internal
# node data frame (ndf)
graph %>%
  get_node_df()
```

---

add_nodes_from_df_cols

Add nodes from distinct values in data frame columns

Description

Add new nodes to a graph object of class dgr_graph using distinct values from one or more columns in a data frame. The values will serve as node labels and the number of nodes added depends on the number of distinct values found in the specified columns.

Usage

```
add_nodes_from_df_cols(graph, df, columns, type = NULL,
  keep_duplicates = FALSE)
```

Arguments

graph	a graph object of class dgr_graph.
df	a data frame from which values will be taken as new nodes for the graph.

`columns` a character vector of column names or a numeric vector of column numbers for the data frame supplied in `df`. The distinct values in these columns will serve as labels for the nodes added to the graph.

`type` an optional, single-length character vector that provides a group identifier for the nodes to be added to the graph.

`keep_duplicates` an option to exclude incoming nodes where the any labels (i.e., values found in columns of the specified `df`) match label values available in the graph's nodes. By default, this is set to `FALSE`.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create an empty graph
graph <- create_graph()

# Create a data frame from
# which several columns have
# values designated as graph nodes
df <-
  data.frame(
    col_1 = c("f", "p", "q"),
    col_2 = c("q", "x", "f"),
    col_3 = c(1, 5, 3),
    col_4 = c("a", "v", "h"),
    stringsAsFactors = FALSE)

# Add nodes from columns `col_1`
# and `col_2` from the data frame
# to the graph object
graph <-
  graph %>%
  add_nodes_from_df_cols(
    df = df,
    columns = c("col_1", "col_2"))

# Show the graph's node data
# frame; duplicate labels are
# prevented with `keep_duplicates =
# FALSE`)
graph %>%
  get_node_df()

# Add new nodes from columns 3 and 4;
# We can specify the columns by their
# numbers as well
graph <-
  graph %>%
```

```
  add_nodes_from_df_cols(
    df = df,
    columns = 3:4)

# Show the graph's node data
# frame; note that nodes didn't
# get made with columns that
# are not character class columns
graph %>%
  get_node_df()
```

`add_nodes_from_table` *Add nodes and attributes to graph from a table*

Description

Add nodes and their attributes to an existing graph object from data in a CSV file or a data frame.

Usage

```
add_nodes_from_table(graph, table, label_col = NULL, type_col = NULL,
  set_type = NULL, drop_cols = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`table`	either a path to a CSV file, or, a data frame object.
`label_col`	an option to apply a column of data in the table as `label` attribute values.
`type_col`	an option to apply a column of data in the table as `type` attribute values.
`set_type`	an optional string to apply a `type` attribute to all nodes created from data in the external table.
`drop_cols`	an optional column selection statement for dropping columns from the external table before inclusion as attributes in the graph's internal node data frame. Several columns can be dropped by name using the syntax `col_1 & col_2 & ...`. Columns can also be dropped using a numeric column range with `:` (e.g., `5:8`), or, by using the `:` between column names to specify the range (e.g., `col_5_name:col_8_name`).

Value

a graph object of class `dgr_graph`.

Examples

```
# To add nodes from the dataset called
# `currencies` (available as a dataset
# in the package), call the
# `add_nodes_from_table()` function
# after creating an empty graph; new
# node ID values will be created as
# monotonically-increasing values
graph_1 <-
  create_graph() %>%
  add_nodes_from_table(
    table = currencies)

# View part of the graph's internal
# node data frame (ndf)
graph_1 %>%
  get_node_df() %>%
  .[, 1:5] %>%
  head()

# If you would like to assign
# any of the table's columns as
# `type` or `label` attributes,
# this can be done with the `type_col`
# and `label_col` arguments; to set
# a static `type` attribute for all
# of the table records, use `set_type`
graph_2 <-
  create_graph() %>%
  add_nodes_from_table(
    table = currencies,
    label_col = iso_4217_code,
    set_type = currency)

# View part of the graph's internal ndf
graph_2 %>%
  get_node_df() %>%
  .[, 1:5] %>%
  head()

# Suppose we would like to not
# include certain columns from the
# external table in the resulting
# graph; we can use the `drop_cols`
# argument to choose which columns
# to not include as attributes
graph_3 <-
  create_graph() %>%
  add_nodes_from_table(
    table = currencies,
    label_col = iso_4217_code,
    set_type = currency,
```

```
    drop_cols = exponent & currency_name)

# Show the node attribute names
# for the graph; note that the
# `exponent` and `currency_name`
# columns are not attributes in the
# graph's internal node data frame
graph_3 %>%
  get_node_df() %>%
  colnames()
```

---

add_node_clones_ws *Add clones of a selection of nodes*

Description

Add new nodes to a graph object of class dgr_graph which are clones of nodes in an active selection of nodes. All node attributes are preserved except for the node label attribute (to maintain the uniqueness of non-NA node label values). A vector of node label can be provided to bind new labels to the cloned nodes.

Usage

```
add_node_clones_ws(graph, add_edges = FALSE, direction = NULL,
  label = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
add_edges	an option for whether to add edges from the selected nodes to each of their clones, or, in the opposite direction.
direction	using from will create new edges from existing nodes to the new, cloned nodes. The to option will create new edges directed toward the existing nodes.
label	an optional vector of node label values. The vector length should correspond to the number of nodes in the active selection of nodes.

Value

a graph object of class dgr_graph.

Examples

```
# Create a graph with a path of
# nodes; supply `label`, `type`,
# and `value` node attributes,
# and select the created nodes
graph <-
  create_graph() %>%
```

```
  add_path(
    n = 3,
    label = c("d", "g", "r"),
    type = c("a", "b", "c")) %>%
  select_last_nodes_created()

# Display the graph's internal
# node data frame
graph %>%
  get_node_df()

# Create clones of all nodes
# in the selection but assign
# new node label values
# (leaving `label` as NULL
# yields NA values)
graph <-
  graph %>%
  add_node_clones_ws(
    label = c("a", "b", "v"))

# Display the graph's internal
# node data frame: nodes `4`,
# `5`, and `6` are clones of
# `1`, `2`, and `3`
graph %>%
  get_node_df()

# Select the last nodes
# created (`4`, `5`, and `6`)
# and clone those nodes and
# their attributes while
# creating new edges between
# the new and existing nodes
graph <-
  graph %>%
  select_last_nodes_created() %>%
  add_node_clones_ws(
    add_edges = TRUE,
    direction = "to",
    label = c("t", "z", "s"))

# Display the graph's internal
# edge data frame; there are
# edges between the selected
# nodes and their clones
graph %>%
  get_edge_df()
```

---

add_node_df *Add nodes from a node data frame to an existing graph object*

Description

With a graph object of class dgr_graph add nodes from a node data frame to that graph.

Usage

```
add_node_df(graph, node_df)
```

Arguments

graph	a graph object of class dgr_graph.
node_df	a node data frame that is created using create_node_df.

Value

a graph object of class dgr_graph.

Examples

```
# Create an empty graph
graph <- create_graph()

# Create a node data frame (ndf)
ndf <-
  create_node_df(n = 2)

# Add the node data frame to
# the graph object to create
# a graph with nodes
graph <-
  graph %>%
  add_node_df(
    node_df = ndf)

# Inspect the graph's ndf
graph %>%
  get_node_df()

# Create another ndf
ndf_2 <-
  create_node_df(n = 3)

# Add the second node data
# frame to the graph object
# to add more nodes with
# attributes to the graph
graph <-
  graph %>%
  add_node_df(
    node_df = ndf_2)

# View the graph's internal
```

```
# node data frame using the
# `get_node_df()` function
graph %>%
  get_node_df()
```

---

add_n_nodes *Add one or several unconnected nodes to the graph*

Description

Add n new nodes to a graph object of class dgr_graph. Optionally, set node type values for the new nodes.

Usage

```
add_n_nodes(graph, n, type = NULL, label = NULL, node_aes = NULL,
  node_data = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
n	the number of new nodes to add to the graph.
type	an optional character vector that provides group identifiers for the nodes to be added.
label	an optional character object that describes the nodes to be added.
node_aes	an optional list of named vectors comprising node aesthetic attributes. The helper function node_aes() is strongly recommended for use here as it contains arguments for each of the accepted node aesthetic attributes (e.g., shape, style, color, fillcolor).
node_data	an optional list of named vectors comprising node data attributes. The helper function node_data() is strongly recommended for use here as it helps bind data specifically to the created nodes.

Value

a graph object of class dgr_graph.

Examples

```
# Create an empty graph and
# add 5 nodes; these nodes
# will be assigned ID values
# from `1` to `5`
graph <-
  create_graph() %>%
  add_n_nodes(n = 5)
```

```
# Get the graph's node IDs
graph %>%
  get_node_ids()
```

---

add_n_nodes_ws *Add a multiple of new nodes with edges to or from one or more selected nodes*

Description

Add n new nodes to or from one or more nodes available as a selection in a graph object of class dgr_graph. New graph edges will all move either from the nodes in the selection toward the newly created nodes (with the option direction = "from"), or to the selected nodes already in the graph (using direction = "to"). Optionally, set node type and edge rel values for all the new nodes and edges created, respectively.

Selections of nodes can be performed using the following select_... functions: select_nodes(), select_last_nodes_created(), select_nodes_by_degree(), select_nodes_by_id(), or select_nodes_in_neighbo
Selections of nodes can also be performed using the following traversal functions: (trav_...): trav_out(), trav_in(), trav_both(), trav_in_node(), trav_out_node().

Usage

```
add_n_nodes_ws(graph, n, direction = NULL, type = NULL, label = NULL,
  rel = NULL, node_aes = NULL, edge_aes = NULL, node_data = NULL,
  edge_data = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
n	the number of new nodes to attach as successor nodes to the nodes in the selection.
direction	using from will create new edges from existing nodes to the new nodes. The to option will create new edges directed toward the existing nodes.
type	an optional character vector that provides group identifiers for the nodes to be added.
label	an optional character object that describes the nodes to be added.
rel	an optional string to apply a rel attribute to all newly created edges.
node_aes	an optional list of named vectors comprising node aesthetic attributes. The helper function node_aes() is strongly recommended for use here as it contains arguments for each of the accepted node aesthetic attributes (e.g., shape, style, color, fillcolor).
edge_aes	an optional list of named vectors comprising edge aesthetic attributes. The helper function edge_aes() is strongly recommended for use here as it contains arguments for each of the accepted edge aesthetic attributes (e.g., shape, style, penwidth, color).

node_data — an optional list of named vectors comprising node data attributes. The helper function node_data() is strongly recommended for use here as it helps bind data specifically to the created nodes.

edge_data — an optional list of named vectors comprising edge data attributes. The helper function edge_data() is strongly recommended for use here as it helps bind data specifically to the created edges.

Value

a graph object of class dgr_graph.

Examples

```
# Create an empty graph, add a node to it, select
# that node, and then add 5 more nodes to the graph
# with edges from the original node to all of the
# new nodes
graph <-
  create_graph() %>%
  add_n_nodes(n = 1) %>%
  select_last_nodes_created() %>%
  add_n_nodes_ws(
    n = 5,
    direction = "from")

# Get the graph's nodes
graph %>%
  get_node_ids()
#> [1] 1 2 3 4 5 6

# Get the graph's edges
graph %>%
  get_edges()
#> "1->2" "1->3" "1->4" "1->5" "1->6"

# Create an empty graph, add a node to it, select
# that node, and then add 5 more nodes to the graph
# with edges toward the original node from all of
# the new nodes
graph <-
  create_graph() %>%
  add_n_nodes(n = 1) %>%
  select_last_nodes_created() %>%
  add_n_nodes_ws(
    n = 5,
    direction = "to")

# Get the graph's nodes
graph %>%
  get_node_ids()
#> [1] 1 2 3 4 5 6
```

```
# Get the graph's edges
graph %>%
  get_edges()
#> "2->1" "3->1" "4->1" "5->1" "6->1"
```

`add_n_node_clones` *Add one or several clones of an existing node to the graph*

Description

Add `n` new nodes to a graph object of class `dgr_graph` which are clones of a node already in the graph. All node attributes are preserved except for the node `label` attribute (to maintain the uniqueness of non-`NA` node label values). A vector of node `label` can be provided to bind new labels to the cloned nodes.

Usage

```
add_n_node_clones(graph, n, node, label = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`n`	the number of node clones to add to the graph.
`node`	a node ID corresponding to the graph node to be cloned.
`label`	an optional vector of node label values. The vector length should correspond to the value set for `n`.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a graph with a path of
# nodes; supply `label`, `type`,
# and `value` node attributes
graph <-
  create_graph() %>%
  add_path(
    n = 3,
    label = c("d", "g", "r"),
    type = c("a", "b", "c"))

# Display the graph's internal
# node data frame
graph %>%
  get_node_df()
```

```
# Create 3 clones of node `1`
# but assign new node label
# values (leaving `label` as
# NULL yields NA values)
graph <-
  graph %>%
  add_n_node_clones(
    n = 3,
    node = 1,
    label = c("x", "y", "z"))

# Display the graph's internal
# node data frame: nodes `4`,
# `5`, and `6` are clones of `1`
graph %>%
  get_node_df()
```

---

add_path *Add a path of nodes to the graph*

Description

With a graph object of class dgr_graph, add a node path to the graph.

Usage

```
add_path(graph, n, type = NULL, label = TRUE, rel = NULL,
  node_aes = NULL, edge_aes = NULL, node_data = NULL, edge_data = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
n	the number of nodes comprising the path.
type	an optional string that describes the entity type for the nodes to be added.
label	either a vector object of length n that provides optional labels for the new nodes, or, a boolean value where setting to TRUE ascribes node IDs to the label and FALSE yields a blank label.
rel	an optional string for providing a relationship label to all new edges created in the node path.
node_aes	an optional list of named vectors comprising node aesthetic attributes. The helper function node_aes() is strongly recommended for use here as it contains arguments for each of the accepted node aesthetic attributes (e.g., shape, style, color, fillcolor).
edge_aes	an optional list of named vectors comprising edge aesthetic attributes. The helper function edge_aes() is strongly recommended for use here as it contains arguments for each of the accepted edge aesthetic attributes (e.g., shape, style, penwidth, color).

node_data — an optional list of named vectors comprising node data attributes. The helper function node_data() is strongly recommended for use here as it helps bind data specifically to the created nodes.

edge_data — an optional list of named vectors comprising edge data attributes. The helper function edge_data() is strongly recommended for use here as it helps bind data specifically to the created edges.

Value

a graph object of class dgr_graph.

Examples

```
# Create a new graph and add
# 2 paths of varying lengths
graph <-
  create_graph() %>%
  add_path(
    n = 4,
    type = "path") %>%
  add_path(
    n = 5,
    type = "path")

# Get node information
# from this graph
graph %>%
  get_node_info()

# Node and edge aesthetic and data
# attributes can be specified in
# the `node_aes`, `edge_aes`,
# `node_data`, and `edge_data`
# arguments

set.seed(23)

graph_w_attrs <-
  create_graph() %>%
  add_path(
    n = 3,
    label = c(
      "one", "two", "three"),
    type = c(
      "a", "a", "b"),
    rel = "A",
    node_aes = node_aes(
      fillcolor = "steelblue"),
    edge_aes = edge_aes(
      color = "red",
      penwidth = 1.2),
    node_data = node_data(
```

```
        value = c(
          1.6, 2.8, 3.4)),
      edge_data = edge_data(
        value =
          rnorm(
            n = 2,
            mean = 5.0,
            sd = 1.0)))

# Get the graph's node data frame
graph_w_attrs %>%
  get_node_df()

# Get the graph's edge data frame
graph_w_attrs %>%
  get_edge_df()
```

---

add_pa_graph *Add a preferential attachment graph*

Description

To an existing graph object, add a graph built according to the Barabasi-Albert model, which uses preferential attachment in its stochastic algorithm.

Usage

```
add_pa_graph(graph, n, m = NULL, power = 1, out_dist = NULL,
  use_total_degree = FALSE, zero_appeal = 1, algo = "psumtree",
  type = NULL, label = TRUE, rel = NULL, node_aes = NULL,
  edge_aes = NULL, node_data = NULL, edge_data = NULL, set_seed = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
n	the number of nodes comprising the preferential attachment graph.
m	the number of edges to add in each time step.
power	the power of the preferential attachment. The default value of 1 indicates a linear preferential attachment.
out_dist	a numeric vector that provides the distribution of the number of edges to add in each time step.
use_total_degree	a logical value (default is TRUE) that governs whether the total degree should be used for calculating the citation probability. If FALSE, the indegree is used.
zero_appeal	a measure of the attractiveness of the nodes with no adjacent edges.

algo	the algorithm to use to generate the graph. The available options are psumtree, psumtree-multiple, and bag. With the psumtree algorithm, a partial prefix-sum tree is used to to create the graph. Any values for power and zero_appeal can be provided and this algorithm never generates multiple edges. The psumtree-multiple algorithm also uses a partial prefix-sum tree but the difference here is that multiple edges are allowed. The bag algorithm places the node IDs into a bag as many times as their in-degree (plus once more). The required number of cited nodes are drawn from the bag with replacement. Multiple edges may be produced using this method (it is not disallowed).
type	an optional string that describes the entity type for all the nodes to be added.
label	a boolean value where setting to TRUE ascribes node IDs to the label and FALSE yields a blank label.
rel	an optional string for providing a relationship label to all edges to be added.
node_aes	an optional list of named vectors comprising node aesthetic attributes. The helper function node_aes() is strongly recommended for use here as it contains arguments for each of the accepted node aesthetic attributes (e.g., shape, style, color, fillcolor).
edge_aes	an optional list of named vectors comprising edge aesthetic attributes. The helper function edge_aes() is strongly recommended for use here as it contains arguments for each of the accepted edge aesthetic attributes (e.g., shape, style, penwidth, color).
node_data	an optional list of named vectors comprising node data attributes. The helper function node_data() is strongly recommended for use here as it helps bind data specifically to the created nodes.
edge_data	an optional list of named vectors comprising edge data attributes. The helper function edge_data() is strongly recommended for use here as it helps bind data specifically to the created edges.
set_seed	supplying a value sets a random seed of the Mersenne-Twister implementation.

Examples

```
# Create an undirected PA
# graph with 100 nodes, adding
# 2 edges at every time step
pa_graph <-
  create_graph(
    directed = FALSE) %>%
  add_pa_graph(
    n = 100,
    m = 1)

# Get a count of nodes
pa_graph %>%
  count_nodes()

# Get a count of edges
pa_graph %>%
  count_edges()
```

add_prism *Add a prism of nodes to the graph*

Description

With a graph object of class dgr_graph, add a node prism to the graph.

Usage

```
add_prism(graph, n, type = NULL, label = TRUE, rel = NULL,
  node_aes = NULL, edge_aes = NULL, node_data = NULL, edge_data = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
n	the number of nodes describing the shape of the prism. For example, the triagonal prism has n equal to 3 and it is composed of 6 nodes and 9 edges. For any n-gonal prism, the graph will be generated with 2n nodes and 3n edges.
type	an optional string that describes the entity type for the nodes to be added.
label	either a vector object of length n that provides optional labels for the new nodes, or, a boolean value where setting to TRUE ascribes node IDs to the label and FALSE yields a blank label.
rel	an optional string for providing a relationship label to all new edges created in the node prism.
node_aes	an optional list of named vectors comprising node aesthetic attributes. The helper function node_aes() is strongly recommended for use here as it contains arguments for each of the accepted node aesthetic attributes (e.g., shape, style, color, fillcolor).
edge_aes	an optional list of named vectors comprising edge aesthetic attributes. The helper function edge_aes() is strongly recommended for use here as it contains arguments for each of the accepted edge aesthetic attributes (e.g., shape, style, penwidth, color).
node_data	an optional list of named vectors comprising node data attributes. The helper function node_data() is strongly recommended for use here as it helps bind data specifically to the created nodes.
edge_data	an optional list of named vectors comprising edge data attributes. The helper function edge_data() is strongly recommended for use here as it helps bind data specifically to the created edges.

Value

a graph object of class dgr_graph.

Examples

```
# Create a new graph and
# add 2 prisms
graph <-
  create_graph() %>%
  add_prism(
    n = 3,
    type = "prism",
    label = "a") %>%
  add_prism(
    n = 3,
    type = "prism",
    label = "b")

# Get node information from this graph
graph %>%
  get_node_info()

# Node and edge aesthetic and data
# attributes can be specified in
# the `node_aes`, `edge_aes`,
# `node_data`, and `edge_data`
# arguments

set.seed(23)

graph_w_attrs <-
  create_graph() %>%
  add_prism(
    n = 3,
    label = c(
      "one", "two",
      "three", "four",
      "five", "six"),
    type = c(
      "a", "a",
      "b", "b",
      "c", "c"),
    rel = "A",
    node_aes = node_aes(
      fillcolor = "steelblue"),
    edge_aes = edge_aes(
      color = "red",
      penwidth = 1.2),
    node_data = node_data(
      value = c(
        1.6, 2.8, 3.4,
        3.2, 5.3, 6.2)),
    edge_data = edge_data(
      value =
        rnorm(
          n = 9,
```

```
          mean = 5.0,
          sd = 1.0)))

# Get the graph's node data frame
graph_w_attrs %>%
  get_node_df()

# Get the graph's edge data frame
graph_w_attrs %>%
  get_edge_df()
```

`add_reverse_edges_ws` *Add new edges in the opposite directions of a selection of edges*

Description

Add edges in the opposite direction of one or more edges available as an edge selection in a graph object of class `dgr_graph`. New graph edges have the opposite edge definitions as those in the selection. For example, a graph with the edge `1->2` in its active selection will gain a new `2->1` edge. There is also the option to assign a common `rel` grouping to the newly created edges. Upon addition of the edges, the edge selection will be retained for further selection or traversal operations.

Selections of edges can be performed using the following `select_...` functions: `select_edges()`, `select_last_edge()`, or `select_edges_by_node_id()`. Selections of edges can also be performed using the following traversal functions: `trav_out_edge()`, `trav_in_edge()`, or `trav_both_edge()`.

Usage

```
add_reverse_edges_ws(graph, rel = NULL, edge_aes = NULL, edge_data = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`rel`	an optional string to apply a `rel` attribute to all newly created edges.
`edge_aes`	an optional list of named vectors comprising edge aesthetic attributes. The helper function `edge_aes()` is strongly recommended for use here as it contains arguments for each of the accepted edge aesthetic attributes (e.g., `shape`, `style`, `penwidth`, `color`).
`edge_data`	an optional list of named vectors comprising edge data attributes. The helper function `edge_data()` is strongly recommended for use here as it helps bind data specifically to the created edges.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create an empty graph, add 2 nodes to it,
# and create the edge `1->2`
graph <-
  create_graph() %>%
  add_n_nodes(
    n = 2,
    type = "type_a",
    label = c("a_1", "a_2")) %>%
  add_edge(
    from = 1,
    to = 2,
    rel = "a")

# Get the graph's edges
graph %>%
  get_edge_ids()

# Select the edge and create 2 additional edges
# with the opposite definition of `1->2`, which
# is `2->1`; also, apply, different `rel` values
# (`b` and `c`)
graph <-
  graph %>%
  select_edges() %>%
  add_reverse_edges_ws(rel = "b") %>%
  add_reverse_edges_ws(rel = "c") %>%
  clear_selection()

# Get the graph's edge data frame
graph %>%
  get_edge_df()
```

---

add_smallworld_graph *Add a Watts-Strogatz small-world graph*

Description

To an existing graph object, add a graph built according to the Watts-Strogatz small-world model, which uses a lattice along with a rewiring probability to randomly modify edge definitions.

Usage

```
add_smallworld_graph(graph, dimension, size, neighborhood, p, loops = FALSE,
  multiple = FALSE, type = NULL, label = TRUE, rel = NULL,
  node_aes = NULL, edge_aes = NULL, node_data = NULL, edge_data = NULL,
  set_seed = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
dimension	the dimension of the starting lattice.
size	the size of the lattice across each dimension.
neighborhood	the neighborhood where the lattice nodes are to be connected.
p	the rewiring probability.
loops	a logical value (default is FALSE) that governs whether loops are allowed to be created.
multiple	a logical value (default is FALSE) that governs whether multiple edges are allowed to be created.
type	an optional string that describes the entity type for all the nodes to be added.
label	a boolean value where setting to TRUE ascribes node IDs to the label and FALSE yields a blank label.
rel	an optional string for providing a relationship label to all edges to be added.
node_aes	an optional list of named vectors comprising node aesthetic attributes. The helper function node_aes() is strongly recommended for use here as it contains arguments for each of the accepted node aesthetic attributes (e.g., shape, style, color, fillcolor).
edge_aes	an optional list of named vectors comprising edge aesthetic attributes. The helper function edge_aes() is strongly recommended for use here as it contains arguments for each of the accepted edge aesthetic attributes (e.g., shape, style, penwidth, color).
node_data	an optional list of named vectors comprising node data attributes. The helper function node_data() is strongly recommended for use here as it helps bind data specifically to the created nodes.
edge_data	an optional list of named vectors comprising edge data attributes. The helper function edge_data() is strongly recommended for use here as it helps bind data specifically to the created edges.
set_seed	supplying a value sets a random seed of the Mersenne-Twister implementation.

Examples

```
# Create an undirected smallworld
# graph with 100 nodes using
# a probability value of 0.05
smallworld_graph <-
  create_graph(
    directed = FALSE) %>%
  add_smallworld_graph(
    dimension = 1,
    size = 50,
    neighborhood = 1,
    p = 0.05,
    set_seed = 23)
```

```
# Get a count of nodes
smallworld_graph %>%
  count_nodes()

# Get a count of edges
smallworld_graph %>%
  count_edges()
```

`add_star` *Add a star of nodes to the graph*

Description

With a graph object of class `dgr_graph`, add a node star to the graph.

Usage

```
add_star(graph, n, type = NULL, label = TRUE, rel = NULL,
  node_aes = NULL, edge_aes = NULL, node_data = NULL, edge_data = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`n`	the number of nodes comprising the star. The first node will be the center of the star.
`type`	an optional string that describes the entity type for the nodes to be added.
`label`	either a vector object of length `n` that provides optional labels for the new nodes, or, a boolean value where setting to `TRUE` ascribes node IDs to the label and `FALSE` yields a blank label.
`rel`	an optional string for providing a relationship label to all new edges created in the node star.
`node_aes`	an optional list of named vectors comprising node aesthetic attributes. The helper function `node_aes()` is strongly recommended for use here as it contains arguments for each of the accepted node aesthetic attributes (e.g., `shape`, `style`, `color`, `fillcolor`).
`edge_aes`	an optional list of named vectors comprising edge aesthetic attributes. The helper function `edge_aes()` is strongly recommended for use here as it contains arguments for each of the accepted edge aesthetic attributes (e.g., `shape`, `style`, `penwidth`, `color`).
`node_data`	an optional list of named vectors comprising node data attributes. The helper function `node_data()` is strongly recommended for use here as it helps bind data specifically to the created nodes.
`edge_data`	an optional list of named vectors comprising edge data attributes. The helper function `edge_data()` is strongly recommended for use here as it helps bind data specifically to the created edges.

Value

a graph object of class dgr_graph.

Examples

```
# Create a new graph and add 2
# stars of varying numbers of nodes
graph <-
  create_graph() %>%
  add_star(
    n = 4,
    type = "four_star") %>%
  add_star(
    n = 5,
    type = "five_star")

# Get node information from this graph
graph %>%
  get_node_info()

# Node and edge aesthetic and data
# attributes can be specified in
# the `node_aes`, `edge_aes`,
# `node_data`, and `edge_data`
# arguments

set.seed(23)

graph_w_attrs <-
  create_graph() %>%
  add_star(
    n = 4,
    label = c(
      "one", "two",
      "three", "four"),
    type = c(
      "a", "a", "b", "b"),
    rel = "A",
    node_aes = node_aes(
      fillcolor = "steelblue"),
    edge_aes = edge_aes(
      color = "red",
      penwidth = 1.2),
    node_data = node_data(
      value = c(
        1.6, 2.8, 3.4, 8.3)),
    edge_data = edge_data(
      value =
        rnorm(
          n = 3,
          mean = 5.0,
          sd = 1.0)))
```

```
# Get the graph's node data frame
graph_w_attrs %>%
  get_node_df()

# Get the graph's edge data frame
graph_w_attrs %>%
  get_edge_df()
```

`clear_selection` *Clear an active selection of nodes or edges*

Description

Clear the selection of nodes or edges within a graph object.

Usage

```
clear_selection(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a graph with
# a single path
graph <-
  create_graph() %>%
  add_path(n = 5)

# Select nodes with IDs `1`
# and `3`
graph <-
  graph %>%
  select_nodes(
    nodes = c(1, 3))

# Verify that a node selection
# has been made
graph %>%
  get_selection()

# Clear the selection with
# `clear_selection()`
```

```
graph <-
  graph %>%
  clear_selection()

# Verify that the node
# selection has been cleared
graph %>%
  get_selection()
```

`colorize_edge_attrs` *Apply colors based on edge attribute values*

Description

Within a graph's internal edge data frame (edf), use a categorical edge attribute to generate a new edge attribute with color values.

Usage

```
colorize_edge_attrs(graph, edge_attr_from, edge_attr_to, cut_points = NULL,
  palette = "Spectral", alpha = NULL, reverse_palette = FALSE,
  default_color = "#D9D9D9")
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`edge_attr_from`	the name of the edge attribute column from which color values will be based.
`edge_attr_to`	the name of the new edge attribute to which the color values will be applied.
`cut_points`	an optional vector of numerical breaks for bucketizing continuous numerical values available in a edge attribute column.
`palette`	can either be: (1) a palette name from the RColorBrewer package (e.g., `Greens`, `OrRd`, `RdYlGn`), (2) `viridis`, which indicates use of the `viridis` color scale from the package of the same name, or (3) a vector of hexadecimal color names.
`alpha`	an optional alpha transparency value to apply to the generated colors. Should be in the range of `0` (completely transparent) to `100` (completely opaque).
`reverse_palette`	an option to reverse the order of colors in the chosen palette. The default is `FALSE`.
`default_color`	a hexadecimal color value to use for instances when the values do not fall into the bucket ranges specified in the `cut_points` vector.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a graph with 5
# nodes and 4 edges
graph <-
  create_graph() %>%
  add_path(n = 5) %>%
  set_edge_attrs(
    edge_attr = weight,
    values = c(3.7, 6.3, 9.2, 1.6))

# We can bucketize values in
# the edge `weight` attribute using
# `cut_points` and, by doing so,
# assign colors to each of the
# bucketed ranges (for values not
# part of any bucket, a gray color
# is assigned by default)
graph <-
  graph %>%
  colorize_edge_attrs(
    edge_attr_from = weight,
    edge_attr_to = color,
    cut_points = c(0, 2, 4, 6, 8, 10),
    palette = "RdYlGn")

# Now there will be a `color`
# edge attribute with distinct
# colors (from the RColorBrewer
# Red-Yellow-Green palette)
graph %>%
  get_edge_df()
```

---

colorize_node_attrs *Apply colors based on node attribute values*

Description

Within a graph's internal node data frame (ndf), use a categorical node attribute to generate a new node attribute with color values.

Usage

```
colorize_node_attrs(graph, node_attr_from, node_attr_to, cut_points = NULL,
  palette = "Spectral", alpha = NULL, reverse_palette = FALSE,
  default_color = "#D9D9D9")
```

Arguments

graph	a graph object of class dgr_graph.
node_attr_from	the name of the node attribute column from which color values will be based.
node_attr_to	the name of the new node attribute to which the color values will be applied.
cut_points	an optional vector of numerical breaks for bucketizing continuous numerical values available in a node attribute column.
palette	can either be: (1) a palette name from the RColorBrewer package (e.g., Greens, OrRd, RdYlGn), (2) viridis, which indicates use of the viridis color scale from the package of the same name, or (3) a vector of hexadecimal color names.
alpha	an optional alpha transparency value to apply to the generated colors. Should be in the range of 0 (completely transparent) to 100 (completely opaque).
reverse_palette	an option to reverse the order of colors in the chosen palette. The default is FALSE.
default_color	a hexadecimal color value to use for instances when the values do not fall into the bucket ranges specified in the cut_points vector.

Value

a graph object of class dgr_graph.

Examples

```
# Create a graph with 8
# nodes and 7 edges
graph <-
  create_graph() %>%
  add_path(n = 8) %>%
  set_node_attrs(
    node_attr = weight,
    values = c(
      8.2, 3.7, 6.3, 9.2,
      1.6, 2.5, 7.2, 5.4))

# Find group membership values for all nodes
# in the graph through the Walktrap community
# finding algorithm and join those group values
# to the graph's internal node data frame (ndf)
# with the `join_node_attrs()` function
graph <-
  graph %>%
  join_node_attrs(
    df = get_cmty_walktrap(.))

# Inspect the number of distinct communities
graph %>%
  get_node_attrs(
    node_attr = walktrap_group) %>%
```

```
  unique() %>%
  sort()

# Visually distinguish the nodes in the different
# communities by applying colors using the
# `colorize_node_attrs()` function; specifically,
# set different `fillcolor` values with an alpha
# value of 90 and apply opaque colors to the node
# border (with the `color` node attribute)
graph <-
  graph %>%
  colorize_node_attrs(
    node_attr_from = walktrap_group,
    node_attr_to = fillcolor,
    palette = "Greens",
    alpha = 90) %>%
  colorize_node_attrs(
    node_attr_from = walktrap_group,
    node_attr_to = color,
    palette = "viridis",
    alpha = 80)

# Show the graph's internal node data frame
graph %>%
  get_node_df()

# Create a graph with 8 nodes and 7 edges
graph <-
  create_graph() %>%
  add_path(n = 8) %>%
  set_node_attrs(
    node_attr = weight,
    values = c(
      8.2, 3.7, 6.3, 9.2,
      1.6, 2.5, 7.2, 5.4))

# We can bucketize values in `weight` using
# `cut_points` and assign colors to each of the
# bucketed ranges (for values not part of any
# bucket, a gray color is assigned by default)
graph <-
  graph %>%
  colorize_node_attrs(
    node_attr_from = weight,
    node_attr_to = fillcolor,
    cut_points = c(1, 3, 5, 7, 9))

# Now there will be a `fillcolor` node attribute
# with distinct colors (the `#D9D9D9` color is
# the default `gray85` color)
graph %>%
  get_node_df()
```

combine_edfs *Combine multiple edge data frames into a single edge data frame*

Description

Combine several edge data frames in the style of `rbind`, except, it works regardless of the number and ordering of the columns.

Usage

```
combine_edfs(...)
```

Arguments

`...`	two or more edge data frames, which contain edge IDs and associated attributes.

Value

a combined edge data frame.

Examples

```
# Create an edge data frame (edf)
edf_1 <-
  create_edge_df(
    from = c(1, 1, 2, 3),
    to = c(2, 4, 4, 1),
    rel = "requires",
    color = "green",
    data = c(2.7, 8.9, 2.6, 0.6))

# Create a second edge data frame
edf_2 <-
  create_edge_df(
    from = c(5, 7, 8, 8),
    to = c(7, 8, 6, 5),
    rel = "receives",
    arrowhead = "dot",
    color = "red")

# Combine the two edge data frames
all_edges <-
  combine_edfs(edf_1, edf_2)

# View the combined edge data frame
all_edges
```

combine_graphs *Combine two graphs into a single graph*

Description

Combine two graphs in order to make a new graph.

Usage

```
combine_graphs(x, y)
```

Arguments

x	a DiagrammeR graph object to which another graph will be unioned. This graph should be considered the graph from which global graph attributes will be inherited in the resulting graph.
y	a DiagrammeR graph object that is to be unioned with the graph supplied as x.

Value

a graph object of class dgr_graph.

Examples

```
# Create a graph with a cycle
# containing 6 nodes
graph_cycle <-
 create_graph() %>%
   add_cycle(n = 6)

# Create a random graph with
# 8 nodes and 15 edges using the
# `add_gnm_graph()` function
graph_random <-
  create_graph() %>%
  add_gnm_graph(
    n = 8,
    m = 15,
    set_seed = 23)

# Combine the two graphs in a
# union operation
combined_graph <-
  combine_graphs(
    graph_cycle,
    graph_random)

# Get the number of nodes in
# the combined graph
```

```
combined_graph %>%
  count_nodes()

# The `combine_graphs()`
# function will renumber
# node ID values in graph `y`
# during the union; this ensures
# that node ID values are unique
combined_graph %>%
  get_node_ids()
```

`combine_ndfs` *Combine multiple node data frames*

Description

Combine several node data frames into a single node data frame.

Usage

```
combine_ndfs(...)
```

Arguments

`...`	two or more node data frames, which contain node IDs and associated attributes.

Value

a combined node data frame.

Examples

```
# Create two node data frames
node_df_1 <-
  create_node_df(
    n = 2,
    type = c("a", "b"),
    label = c("D", "Z"),
    value = c(8.4, 3.4))

node_df_2 <-
  create_node_df(
    n = 2,
    type = c("b", "c"),
    label = c("U", "A"),
    value = c(0.4, 3.4))

# Combine the ndfs using the
# `combine_ndfs()` function
node_df_combined <-
```

```
  combine_ndfs(
    node_df_1,
    node_df_2)

# Inspect the combined ndf
node_df_combined
```

---

copy_edge_attrs *Copy an edge attribute column and set the name*

Description

Within a graph's internal edge data frame (edf), copy the contents an existing edge attribute and create a distinct edge attribute within the edf with a different attribute name.

Usage

```
copy_edge_attrs(graph, edge_attr_from, edge_attr_to)
```

Arguments

graph	a graph object of class dgr_graph.
edge_attr_from	the name of the edge attribute column from which values will be copied.
edge_attr_to	the name of the new edge attribute column to which the copied values will be placed.

Value

a graph object of class dgr_graph.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 5,
    m = 8,
    set_seed = 23) %>%
  set_edge_attrs(
    edge_attr = color,
    values = "green")

# Get the graph's internal
# edf to show which edge
# attributes are available
graph %>%
  get_edge_df()
```

```
# Make a copy the `color`
# edge attribute as the
# `color_2` edge attribute
graph <-
  graph %>%
  copy_edge_attrs(
    edge_attr_from = color,
    edge_attr_to = color_2)

# Get the graph's internal
# edf to show that the edge
# attribute had been copied
graph %>%
  get_edge_df()
```

`copy_node_attrs` *Copy a node attribute column and set the name*

Description

Within a graph's internal node data frame (ndf), copy the contents an existing node attribute and create a distinct node attribute within the ndf with a different attribute name.

Usage

```
copy_node_attrs(graph, node_attr_from, node_attr_to)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`node_attr_from`	the name of the node attribute column from which values will be copied.
`node_attr_to`	the name of the new node attribute column to which the copied values will be placed.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 5,
    m = 10,
    set_seed = 23) %>%
```

```
  set_node_attrs(
    node_attr = shape,
    values = "circle") %>%
  set_node_attrs(
    node_attr = value,
    values = rnorm(
      n = count_nodes(.),
      mean = 5,
      sd = 1) %>% round(1))

# Get the graph's internal
# ndf to show which node
# attributes are available
graph %>%
  get_node_df()

# Make a copy the `value`
# node attribute as the
# `width` node attribute
graph <-
  graph %>%
  copy_node_attrs(
    node_attr_from = value,
    node_attr_to = size)

# Get the graph's internal
# ndf to show that the node
# attribute had been copied
graph %>%
  get_node_df()
```

`count_asymmetric_node_pairs` *Get the number of asymmetrically-connected node pairs*

Description

Get the number of asymmetrically-connected node pairs. This works for directed graphs.

Usage

```
count_asymmetric_node_pairs(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a single numeric value representing the number of asymmetrically-connected node pairs.

Examples

```
# Create a cycle graph
graph <-
  create_graph() %>%
  add_cycle(n = 5)

# Get a count of asymmetrically-
# connected node pairs
graph %>%
  count_asymmetric_node_pairs()

# Create a full graph and then
# count the asymmetrically-
# connected node pairs
create_graph() %>%
  add_full_graph(n = 10) %>%
  count_asymmetric_node_pairs()
```

---

count_automorphisms *Get the number of automorphisms*

Description

Get the number of automorphisms the graph contains. An automorphism of a graph is a form of symmetry in which the graph is mapped onto itself while preserving edge-node connectivity.

Usage

```
count_automorphisms(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a single numeric value representing the number of automorphisms the graph contains.

Examples

```
# Create a cycle graph
graph <-
  create_graph() %>%
  add_cycle(n = 5)

# Get a count of automorphisms
graph %>%
  count_automorphisms()
```

```
# Create a full graph and then
# count the automorphisms
create_graph() %>%
  add_full_graph(n = 10) %>%
  count_automorphisms()
```

`count_edges` *Get a count of all edges*

Description

From a graph object of class `dgr_graph`, get a count of edges in the graph.

Usage

```
count_edges(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a single-length numeric vector.

Examples

```
# Create a graph with a
# path of nodes and 3
# unconnected nodes
graph <-
  create_graph() %>%
  add_path(n = 3) %>%
  add_n_nodes(n = 3)

# Get a count of all edges
# in the graph
graph %>%
  count_edges()
```

count_graphs_in_graph_series
Count graphs in a graph series object

Description

Counts the total number of graphs in a graph series object.

Usage

```
count_graphs_in_graph_series(graph_series)
```

Arguments

graph_series	a graph series object of type dgr_graph_1D

Value

a numeric vector representing a count of graphs in a graph series object.

Examples

```
# Create three graphs
graph_1 <-
  create_graph() %>%
  add_path(n = 4)

graph_2 <-
  create_graph() %>%
  add_cycle(n = 5)

graph_3 <-
  create_graph() %>%
  add_star(n = 6)

# Create an empty graph series
# and add the graphs
series <-
  create_graph_series() %>%
  add_graph_to_graph_series(
    graph = graph_1) %>%
  add_graph_to_graph_series(
    graph = graph_2) %>%
  add_graph_to_graph_series(
    graph = graph_3)

# Count the number of graphs
# in the graph series
series %>%
  count_graphs_in_graph_series()
```

count_loop_edges *Get count of all loop edges*

Description

From a graph object of class dgr_graph, get a count of all loop edges in the graph.

Usage

```
count_loop_edges(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a numeric vector of single length.

Examples

```
# Create an undirected, full graph
# with 3 nodes and all possible
# edges, including loop edges
graph <-
  create_graph(
    directed = FALSE) %>%
  add_full_graph(
    n = 3,
    keep_loops = TRUE)

# Get a count of all loop edges
# in the graph
graph %>%
  count_loop_edges()
```

count_mutual_node_pairs *Get the number of mutually-connected node pairs*

Description

Get the number of mutually-connected node pairs. This works for directed graphs.

Usage

```
count_mutual_node_pairs(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a single numeric value representing the number of mutually- connected node pairs.

Examples

```
# Create a cycle graph
graph <-
  create_graph() %>%
  add_cycle(n = 5)

# Get a count of mutually-connected
# node pairs
graph %>%
  count_mutual_node_pairs()

# Create a full graph and then
# count the mutually-connected
# node pairs
create_graph() %>%
  add_full_graph(n = 10) %>%
  count_mutual_node_pairs()
```

---

count_nodes *Get a count of all nodes*

Description

From a graph object of class dgr_graph, get a count of nodes in the graph.

Usage

```
count_nodes(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a numeric vector of single length.

Examples

```
# Create a graph with a
# path of nodes and 3
# unconnected nodes
graph <-
  create_graph() %>%
  add_path(n = 3) %>%
  add_n_nodes(n = 3)

# Get a count of all nodes
# in the graph
graph %>%
  count_nodes()
```

`count_s_connected_cmpts` *Get the number of strongly-connected components*

Description

Get the number of strongly-connected components in the graph.

Usage

```
count_s_connected_cmpts(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a single integer value representing the number of strongly-connected graph components.

Examples

```
# Create a graph and add
# several graph islands
graph <-
  create_graph() %>%
  add_islands_graph(
    n_islands = 4,
    island_size = 10,
    p = 1/5,
    edges_between = 1,
    set_seed = 23)

# Get a count of strongly-connected
# components in the graph
```

```
graph %>%
  count_s_connected_cmpts()
```

`count_unconnected_nodes` *Get count of all unconnected nodes*

Description

From a graph object of class `dgr_graph`, get a count of nodes in the graph that are not connected to any other node.

Usage

```
count_unconnected_nodes(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a numeric vector of single length.

Examples

```
# Create a graph with a
# path of nodes and 3
# unconnected nodes
graph <-
  create_graph() %>%
  add_path(n = 3) %>%
  add_n_nodes(n = 3)

# Get a count of all nodes
# in the graph
graph %>%
  count_nodes()

# Get a count of all
# unconnected nodes in the
# graph
graph %>%
  count_unconnected_nodes()
```

count_unconnected_node_pairs

Get the number of unconnected node pairs

Description

Get the number of unconnected node pairs. This works for directed graphs.

Usage

```
count_unconnected_node_pairs(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a single numeric value representing the number of unconnected node pairs.

Examples

```
# Create a cycle graph
graph <-
  create_graph() %>%
  add_cycle(n = 5)

# Get a count of unconnected node
# pairs in the graph
graph %>%
  count_unconnected_node_pairs()

# Create a full graph and then
# count all unconnected node pairs
create_graph() %>%
  add_full_graph(n = 10) %>%
  count_unconnected_node_pairs()
```

count_w_connected_cmpts

Get the number of weakly-connected components

Description

Get the number of weakly-connected components in the graph.

Usage

```
count_w_connected_cmpts(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a single integer value representing the number of weakly-connected graph components.

Examples

```
# Create a cycle graph
graph <-
  create_graph() %>%
  add_cycle(n = 5) %>%
  add_cycle(n = 5)

# Get a count of weakly-connected
# components in the graph
graph %>%
  count_w_connected_cmpts()
```

---

create_edge_df *Create an edge data frame*

Description

Combine several vectors for edges and their attributes into a data frame, which can be combined with other similarly-generated data frames, or, added to a graph object. An edge data frame, or edf, has at least the following columns:

- id (of type integer)

- from (of type integer)

- to (of type integer)

- rel (of type character)

An arbitrary number of additional columns containing aesthetic or data attributes can be part of the edf, so long as they follow the aforementioned columns.

Usage

```
create_edge_df(from, to, rel = NULL, ...)
```

Arguments

from	a vector of node ID values from which edges are outbound. The vector length must equal that of the to vector.
to	a vector of node ID values to which edges are incoming. The vector length must equal that of the from vector.
rel	an optional rel label for each edge.
...	one or more vectors for associated edge attributes.

Value

an edge data frame (edf).

Examples

```
# Create a simple edge data frame (edf) and
# view the results
edf <-
  create_edge_df(
    from = c(1, 2, 3),
    to = c(4, 3, 1),
    rel = "a")

# Display the edge data frame
edf

# Create an edf with additional edge
# attributes (where their classes will
# be inferred from the input vectors)
edf <-
  create_edge_df(
    from = c(1, 2, 3),
    to = c(4, 3, 1),
    rel = "a",
    length = c(50, 100, 250),
    color = "green",
    width = c(1, 5, 2))

# Display the edge data frame
edf
```

---

create_graph *Create a graph object*

Description

Generates a graph object with the option to use node data frames (ndfs) and/or edge data frames (edfs) to populate the initial graph.

Usage

```
create_graph(nodes_df = NULL, edges_df = NULL, directed = TRUE,
  graph_name = NULL, attr_theme = "default", write_backups = FALSE)
```

Arguments

nodes_df	an optional data frame containing, at minimum, a column (called id) which contains node IDs for the graph. Additional columns (node attributes) can be included with values for the named node attribute.
edges_df	an optional data frame containing, at minimum, two columns (called from and to) where node IDs are provided. Additional columns (edge attributes) can be included with values for the named edge attribute.
directed	with TRUE (the default) or FALSE, either directed or undirected edge operations will be generated, respectively.
graph_name	an optional string for labeling the graph object.
attr_theme	the theme (i.e., collection of graph, node, and edge global graph attributes) to use for this graph. The default theme is called default. If this is set to NULL then no global graph attributes will be applied to the graph upon creation.
write_backups	an option to write incremental backups of changing graph states to disk. If TRUE, a subdirectory within the working directory will be created and used to store RDS files. The default value is FALSE so one has to opt in to use this functionality.

Value

a graph object of class dgr_graph.

Examples

```
# With `create_graph()` we can
# simply create an empty graph (and
# add in nodes and edges later
# with other functions)
graph <- create_graph()

# A graph can be created with
# nodes and without having any edges;
# this can be done in 2 steps:
# 1. create a node data frame (ndf)
#    using `create_node_df()`
ndf <-
  create_node_df(n = 4)

# 2. create a new graph object with
#    `create_graph()` and then pass
#    in the ndf to `nodes_df`
graph <-
  create_graph(
    nodes_df = ndf)
```

```
# Get information on the graph's nodes
graph %>%
  get_node_info()

# You can create a similar graph with
# just nodes but also providing a
# range of attributes for the nodes
# (e.g., types, labels, or arbitrary
# 'values')
ndf <-
  create_node_df(
    n = 4,
    label = TRUE,
    type = c("type_1", "type_1",
             "type_5", "type_2"),
    shape = c("circle", "circle",
              "rectangle", "rectangle"),
    values = c(3.5, 2.6, 9.4, 2.7))

graph <-
  create_graph(nodes_df = ndf)

# Get information on the graph's
# internal node data frame (ndf)
graph %>%
  get_node_df()

# A graph can also be created by
# specifying both the nodes and
# edges; create an edge data frame
# (edf) using the `create_edge_df()`
# function:
edf <-
  create_edge_df(
    from = c(1, 2, 3),
    to = c(4, 3, 1),
    rel = "leading_to",
    values = c(7.3, 2.6, 8.3))

# 2. create the graph object with
#    `create_graph()` and pass in
#    the ndf and edf objects
graph <-
  create_graph(
    nodes_df = ndf,
    edges_df = edf)

# Get information on the graph's
# internal edge data frame (edf)
graph %>%
  get_edge_df()

# Get information on the graph's
```

```
# internal node data frame (ndf)
graph %>%
  get_node_df()
```

`create_graph_series` *Create a graph series object*

Description

Create a graph series object for the storage of multiple graphs across a sequential or temporal one-dimensional array.

Usage

```
create_graph_series(graph = NULL, series_name = NULL,
  series_type = "sequential")
```

Arguments

`graph`	a graph object to add to the new graph series object.
`series_name`	an optional name to ascribe to the series.
`series_type`	either a `sequential` type (the default) or a `temporal` type (which requires date-time strings and time zone codes to be supplied).

Value

a graph series object of type `dgr_graph_1D`.

Examples

```
# Create three graphs
graph_1 <-
  create_graph() %>%
  add_path(n = 4)

graph_2 <-
  create_graph() %>%
  add_cycle(n = 5)

graph_3 <-
  create_graph() %>%
  add_star(n = 6)

# Create an empty graph series
# and add the graphs
series <-
  create_graph_series() %>%
  add_graph_to_graph_series(
    graph = graph_1) %>%
```

```
  add_graph_to_graph_series(
    graph = graph_2) %>%
  add_graph_to_graph_series(
    graph = graph_3)

# Count the number of graphs
# in the graph series
series %>%
  count_graphs_in_graph_series()
```

`create_node_df` *Create a node data frame*

Description

Combine several vectors for nodes and their attributes into a data frame, which can be combined with other similarly-generated data frames, or, added to a graph object. A node data frame, or ndf, has at least the following columns:

- `id` (of type `integer`)

- `type` (of type `character`)

- `label` (of type `character`)

An arbitrary number of additional columns containing aesthetic or data attributes can be part of the ndf, so long as they follow the aforementioned columns.

Usage

```
create_node_df(n, type = NULL, label = NULL, ...)
```

Arguments

`n`	the total number of nodes to include in the node data frame.
`type`	an optional `type` for each node.
`label`	an optional `label` for each node.
`...`	one or more vectors for associated node attributes.

Value

a node data frame (ndf).

Examples

```
# Create a node data frame (ndf) where the labels
# are equivalent to the node ID values (this is not
# recommended); the `label` and `type` node
# attributes will always be a `character` class
# whereas `id` will always be an `integer`
node_df <-
```

```
  create_node_df(
    n = 4,
    type = c("a", "a", "b", "b"),
    label = TRUE)

# Display the node data frame
node_df

# Create an ndf with distinct labels and
# additional node attributes (where their classes
# will be inferred from the input vectors)
node_df <-
  create_node_df(
    n = 4,
    type = "a",
    label = c(2384, 3942, 8362, 2194),
    style = "filled",
    color = "aqua",
    shape = c("circle", "circle",
              "rectangle", "rectangle"),
    value = c(3.5, 2.6, 9.4, 2.7))

# Display the node data frame
node_df
```

`currencies` *ISO-4217 currency data.*

Description

A dataset containing currency information from the ISO-4217 standard.

Usage

```
currencies
```

Format

A data frame with 171 rows and 4 variables:

iso_4217_code the three-letter currency code according to the ISO-4217 standard

curr_number the three-digit code number assigned to each currency under the ISO-4217 standard

exponent the base 10 exponent of the minor currency unit in relation to the major currency unit (it can be assumed also to be number of decimal places that is commonly considered for the currency)

currency_name the English name of the currency

Source

https://en.wikipedia.org/wiki/ISO_4217

delete_cache *Delete vectors cached in a graph object*

Description

Delete vectors cached in a graph object of class dgr_graph.

Usage

```
delete_cache(graph, name = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
name	one or more name of vector objects to delete from the cache. If none supplied, all cached vectors available in the graph will be deleted.

Value

a vector.

Examples

```
# Create an empty graph
graph <-
  create_graph()

# Cache 3 different vectors inside
# the graph object
graph <-
  graph %>%
  set_cache(
    name = "a",
    to_cache = 1:4) %>%
  set_cache(
    name = "b",
    to_cache = 5:9) %>%
  set_cache(
    name = "c",
    to_cache = 10:14)

# Delete cache `b`
graph <-
  graph %>%
  delete_cache(name = "b")

# Delete remaining cached vectors
graph <-
  graph %>%
  delete_cache()
```

delete_edge *Delete an edge from an existing graph object*

Description

From a graph object of class dgr_graph, delete an existing edge by specifying either: (1) a pair of node IDs corresponding to the edge (keeping into consideration the direction of the edge in a directed graph), or (2) an edge ID.

Usage

```
delete_edge(graph, from = NULL, to = NULL, id = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
from	a node ID from which the edge to be removed is outgoing. If an edge ID is provided to id, then this argument is ignored. There is the option to use a node label value here (and this must correspondingly also be done for the to argument) for defining node connections. Note that this is only possible if all nodes have distinct label values set and none exist as an empty string.
to	a node ID to which the edge to be removed is incoming. If an edge ID is provided to id, then this argument is ignored. There is the option to use a node label value here (and this must correspondingly also be for the from argument) for defining node connections. Note that this is only possible if all nodes have distinct label values set and none exist as an empty string.
id	an edge ID of the edge to be removed.

Value

a graph object of class dgr_graph.

Examples

```
# Create a graph with 2 nodes
graph <-
  create_graph() %>%
  add_n_nodes(n = 2)

# Add an edge
graph <-
  graph %>%
  add_edge(
    from = 1,
    to = 2)

# Delete the edge
```

```
graph <-
  graph %>%
  delete_edge(
    from = 1,
    to = 2)

# Get the count of edges in the graph
graph %>%
  count_edges()

# Create an undirected graph with
# 2 nodes and an edge
graph_undirected <-
  create_graph(directed = FALSE) %>%
  add_n_nodes(n = 2) %>%
  add_edge(
    from = 1,
    to = 2)

# Delete the edge; the order of node ID
# values provided in `from` and `to`
# don't matter for the undirected case
graph_undirected %>%
  delete_edge(
    from = 2,
    to = 1) %>%
  count_edges()

# The undirected graph has a single
# edge with ID `1`; it can be
# deleted by specifying `id`
graph_undirected %>%
  delete_edge(id = 1) %>%
  count_edges()

# Create a directed graph with 2
# labeled nodes and an edge
graph_labeled_nodes <-
  create_graph() %>%
  add_n_nodes(
    n = 2,
    label = c("one", "two")) %>%
  add_edge(
    from = "one",
    to = "two")

# Delete the edge using the node
# labels in `from` and `to`; this
# is analogous to creating the
# edge using node labels
graph_labeled_nodes %>%
  delete_edge(
    from = "one",
```

```
    to = "two") %>%
  count_edges()
```

`delete_edges_ws` *Delete all selected edges in an edge selection*

Description

In a graph object of class `dgr_graph`, delete all edges present in a selection.

Selections of edges can be performed using the following `select_...` functions: `select_edges()`, `select_last_edge()`, or `select_edges_by_node_id()`. Selections of edges can also be performed using the following traversal functions: `trav_out_edge()`, `trav_in_edge()`, or `trav_both_edge()`.

Usage

```
delete_edges_ws(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a graph
graph <-
  create_graph() %>%
  add_n_nodes(n = 3) %>%
  add_edges_w_string(
    edges = "1->3 1->2 2->3")

# Select edges attached to
# node with ID `3` (these are
# `1`->`3` and `2`->`3`)
graph <-
  graph %>%
  select_edges_by_node_id(nodes = 3)

# Delete edges in selection
graph <-
  graph %>%
  delete_edges_ws()

# Get a count of edges in the graph
graph %>%
  count_edges()
```

`delete_global_graph_attrs` *Delete one of the global graph attributes stored within a graph object*

Description

Delete one of the global attributes stored within a graph object of class `dgr_graph`).

Usage

```
delete_global_graph_attrs(graph, attr = NULL, attr_type = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`attr`	the name of the attribute to delete for the `type` of global attribute specified.
`attr_type`	the specific type of global graph attribute to delete. The type is specified with `graph`, `node`, or `edge`.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a new graph and add
# some extra global graph attrs
graph <-
  create_graph() %>%
  add_global_graph_attrs(
    attr = "overlap",
    value = "true",
    attr_type = "graph") %>%
  add_global_graph_attrs(
    attr = "penwidth",
    value = 3,
    attr_type = "node") %>%
  add_global_graph_attrs(
    attr = "penwidth",
    value = 3,
    attr_type = "edge")

# Inspect the graph's global
# attributes
graph %>%
  get_global_graph_attr_info()

# Delete the `penwidth` attribute
# for the graph's nodes using the
```

```
# `delete_global_graph_attrs()` fcn
graph <-
  graph %>%
  delete_global_graph_attrs(
    attr = "penwidth",
    attr_type = "node")

# View the remaining set of global
# attributes for the graph
graph %>%
  get_global_graph_attr_info()
```

`delete_graph_actions` *Delete one or more graph actions stored within a graph object*

Description

Delete one or more graph actions stored within a graph object of class `dgr_graph`).

Usage

```
delete_graph_actions(graph, actions)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`actions`	either a vector of integer numbers indicating which actions to delete (based on `action_index` values), or, a character vector corresponding to `action_name` values.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 5,
    m = 8,
    set_seed = 23)

# Add three graph actions to the
# graph
graph <-
  graph %>%
```

```
  add_graph_action(
    fcn = "set_node_attr_w_fcn",
    node_attr_fcn = "get_pagerank",
    column_name = "pagerank",
    action_name = "get_pagerank") %>%
  add_graph_action(
    fcn = "rescale_node_attrs",
    node_attr_from = "pagerank",
    node_attr_to = "width",
    action_name = "pagerank_to_width") %>%
  add_graph_action(
    fcn = "colorize_node_attrs",
    node_attr_from = "width",
    node_attr_to = "fillcolor",
    action_name = "pagerank_fillcolor")

# View the graph actions for the graph
# object by using the `get_graph_actions()`
# function
graph %>%
  get_graph_actions()

# Delete the second and third graph
# actions using `delete_graph_actions()`
graph <-
  graph %>%
  delete_graph_actions(
    actions = c(2, 3))

# Verify that these last two graph
# actions were deleted by again using
# the `get_graph_actions()` function
graph %>%
  get_graph_actions()
```

`delete_loop_edges_ws` *Delete all loop edges associated with a selection of nodes*

Description

With a selection of nodes in a graph, remove any associated loop edges.

Usage

```
delete_loop_edges_ws(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a graph object of class dgr_graph.

Examples

```
# Create an undirected, full graph
# of 5 nodes with loops retained
graph <-
  create_graph(
    directed = FALSE) %>%
  add_full_graph(
    n = 5,
    keep_loops = TRUE)

# Select nodes `3` and `4`
# and remove the loop edges
# associated with those nodes
graph <-
  graph %>%
  select_nodes_by_id(
    nodes = 3:4) %>%
  delete_loop_edges_ws()

# Count the number of loop
# edges remaining in the graph
graph %>%
  count_loop_edges()
```

---

delete_node *Delete a node from an existing graph object*

Description

From a graph object of class dgr_graph, delete an existing node by specifying its node ID.

Usage

```
delete_node(graph, node)
```

Arguments

graph	a graph object of class dgr_graph.
node	a node ID for the node to be deleted from the graph.

Value

a graph object of class dgr_graph.

Examples

```
# Create a graph with 5 nodes and
# edges between each in a path
graph <-
  create_graph() %>%
  add_path(n = 5)

# Delete node with ID `3`
graph <- delete_node(graph, node = 3)

# Verify that the node with ID `3`
# is no longer in the graph
graph %>%
  get_node_ids()

# Also note that edges are removed
# since there were edges between the
# removed node to and from other nodes
graph %>%
  get_edges()
```

---

delete_nodes_ws *Delete all selected nodes in a node selection*

Description

In a graph object of class dgr_graph, delete all nodes present in a selection.

Selections of nodes can be performed using the following select_... functions: select_nodes(), select_last_nodes_created(), select_nodes_by_degree(), select_nodes_by_id(), or select_nodes_in_neighbo
Selections of nodes can also be performed using the following traversal functions: (trav_...): trav_out(), trav_in(), trav_both(), trav_in_node(), trav_out_node().

Usage

```
delete_nodes_ws(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a graph object of class dgr_graph.

Examples

```
# Create a graph with 3 nodes
graph <-
  create_graph() %>%
  add_n_nodes(n = 3) %>%
  add_edges_w_string(
    edges = "1->3 1->2 2->3")

# Select node with ID `1`
graph <-
  graph %>%
  select_nodes_by_id(nodes = 1)

# Delete node in selection (this
# also deletes any attached edges)
graph <-
  graph %>%
  delete_nodes_ws()

# Get a count of nodes in the graph
graph %>%
  count_nodes()
```

`deselect_edges` *Deselect any selected edges in a graph*

Description

Deselect edges in a graph object of class `dgr_graph`.

Usage

```
deselect_edges(graph, edges)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`edges`	a vector of edge IDs that should be deselected.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a graph with
# a single path
graph <-
  create_graph() %>%
  add_path(n = 5)

# Select edges with IDs `1`
# and `3`
graph <-
  graph %>%
  select_edges_by_edge_id(
    edges = c(1, 3))

# Verify that an edge selection
# has been made
graph %>%
  get_selection()

# Deselect edge `1`
graph <-
  graph %>%
  select_edges_by_edge_id(
    edges = c(1, 3)) %>%
  deselect_edges(edges = 1)

# Verify that the edge selection
# has been made for edges `1` and
# `3` and that edge `1` has been
# deselected (leaving only `3`)
graph %>%
  get_selection()
```

---

deselect_nodes *Deselect any selected nodes in a graph*

Description

Deselect nodes in a graph object of class dgr_graph.

Usage

```
deselect_nodes(graph, nodes)
```

Arguments

graph	a graph object of class dgr_graph.
nodes	a vector of node IDs that should be deselected.

Value

a graph object of class dgr_graph.

Examples

```
# Create a node data frame (ndf)
ndf <-
  create_node_df(
    n = 4,
    type = c("a", "a", "z", "z"),
    label = TRUE,
    value = c(3.5, 2.6, 9.4, 2.7))

# Create an edge data frame (edf)
edf <-
  create_edge_df(
    from = c(1, 2, 3),
    to = c(4, 3, 1),
    rel = c("a", "z", "a"))

# Create a graph with the ndf and edf
graph <-
  create_graph(
    nodes_df = ndf,
    edges_df = edf)

# Explicitly select nodes `1` and `3`
graph <-
  graph %>%
  select_nodes(nodes = c(1, 3)) %>%
  deselect_nodes(nodes = 1)

# Verify that the node selection
# has been made for nodes `1` and
# `3` and that node `1` has been
# deselected (leaving only `3`)
graph %>%
  get_selection()
```

DiagrammeR *R + mermaid.js*

Description

Make diagrams in R using viz.js or mermaid.js with infrastructure provided by htmlwidgets.

Usage

```
DiagrammeR(diagram = "", type = "mermaid", ...)
```

Arguments

diagram	diagram in graphviz or mermaid format or a file (as a connection or file name) containing a diagram specification. The recommended filename extensions are .gv and .mmd for the Graphviz and the mermaid diagram specifications, respectively. If no diagram is provided (diagram = "") then the function will assume that a diagram will be provided by tags and DiagrammeR is just being used for dependency injection.
type	string - either mermaid (default) or grViz indicating the type of diagram spec and the desired parser/renderer
...	any other parameters to pass to grViz or mermaid

Value

An object of class htmlwidget that will intelligently print itself into HTML in a variety of contexts including the R console, within R Markdown documents, and within Shiny output bindings.

Examples

```
## Not run:
# note the whitespace is not important
DiagrammeR("
  graph LR
    A-->B
    A-->C
    C-->E
    B-->D
    C-->D
    D-->F
    E-->F
")

DiagrammeR("
   graph TB
   A-->B
   A-->C
   C-->E
   B-->D
   C-->D
   D-->F
   E-->F
")

DiagrammeR("graph LR;A(Rounded)-->B[Squared];B-->C{A Decision};
 C-->D[Square One];C-->E[Square Two];
 style A fill:#E5E25F;  style B fill:#87AB51; style C fill:#3C8937;
 style D fill:#23772C;  style E fill:#B6E6E6;"
)

# Load in the 'mtcars' dataset
data(mtcars)
connections <- sapply(
```

```
 1:ncol(mtcars)
  ,function(i) {
     paste0(
        i
      ,"(",colnames(mtcars)[i],")---"
      ,i,"-stats("
      ,paste0(
        names(summary(mtcars[,i]))
        ,": "
        ,unname(summary(mtcars[,i]))
        ,collapse="<br/>"
      )
      ,")"
   )
 }
)

DiagrammeR(
   paste0(
     "graph TD;", "\n",
     paste(connections, collapse = "\n"),"\n",
     "classDef column fill:#0001CC, stroke:#0D3FF3, stroke-width:1px;" ,"\n",
     "class ", paste0(1:length(connections), collapse = ","), " column;"
   )
 )

# also with DiagrammeR() you can use tags from htmltools
# just make sure to use class = "mermaid"
library(htmltools)
diagramSpec = "
graph LR;
  id1(Start)-->id2(Stop);
  style id1 fill:#f9f,stroke:#333,stroke-width:4px;
  style id2 fill:#ccf,stroke:#f66,stroke-width:2px,stroke-dasharray: 5, 5;
"
html_print(tagList(
  tags$h1("R + mermaid.js = Something Special")
  ,tags$pre(diagramSpec)
  ,tags$div(class="mermaid",diagramSpec)
  ,DiagrammeR()
))

# sequence diagrams
# Using this "How to Draw a Sequence Diagram"
#   http://www.cs.uku.fi/research/publications/reports/A-2003-1/page91.pdf
# draw some sequence diagrams with DiagrammeR

library(DiagrammeR)

DiagrammeR("
sequenceDiagram;
   customer->>ticket seller: ask for ticket;
   ticket seller->>database: seats;
```

```
   alt tickets available
     database->>ticket seller: ok;
     ticket seller->>customer: confirm;
     customer->>ticket seller: ok;
     ticket seller->>database: book a seat;
     ticket seller->>printer: print ticket;
   else sold out
     database->>ticket seller: none left;
     ticket seller->>customer:  sorry;
   end
")

## End(Not run)
```

`DiagrammeROutput` *Widget output function for use in Shiny*

Description

Widget output function for use in Shiny

Usage

```
DiagrammeROutput(outputId, width = "100%", height = "auto")
```

Arguments

`outputId`	output variable to read from
`width`	a valid CSS unit for the width or a number, which will be coerced to a string and have `px` appended.
`height`	a valid CSS unit for the height or a number, which will be coerced to a string and have `px` appended.

`display_metagraph` *Display a property graph's underlying model*

Description

With a graph object of class `dgr_graph` that is also a property graph (i.e., all nodes have an assigned `type` value and all edges have an assigned `rel` value), display its metagraph in the RStudio Viewer. This representation provides all combinations of edges of different `rel` values to all nodes with distinct `type` values, including any edges to nodes of the same `type` (shown as loops). The pre-condition of the graph being a property graph can be verified by using the `is_property_graph()` function.

Usage

```
display_metagraph(graph)
```

Arguments

graph	a graph object of class dgr_graph. This graph must fulfill the condition of being a property graph, otherwise the function yields an error.

Examples

```
## Not run:
# Create a randomized property
# graph with 1000 nodes and 1350 edges
property_graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 1000,
    m = 1350,
    set_seed = 23) %>%
  select_nodes_by_degree(
    expressions = "deg >= 3") %>%
  set_node_attrs_ws(
    node_attr = type,
    value = "a") %>%
  clear_selection() %>%
  select_nodes_by_degree(
    expressions = "deg < 3") %>%
  set_node_attrs_ws(
    node_attr = type,
    value = "b") %>%
  clear_selection() %>%
  select_nodes_by_degree(
    expressions = "deg == 0") %>%
  set_node_attrs_ws(
    node_attr = type,
    value = "c") %>%
  set_node_attr_to_display(
    attr = type) %>%
  select_edges_by_node_id(
    nodes =
      get_node_ids(.) %>%
      sample(
        size = 0.15 * length(.) %>%
          floor())) %>%
  set_edge_attrs_ws(
    edge_attr = rel,
    value = "r_1") %>%
  invert_selection() %>%
  set_edge_attrs_ws(
    edge_attr = rel,
    value = "r_2") %>%
  clear_selection() %>%
```

```
  copy_edge_attrs(
    edge_attr_from = rel,
    edge_attr_to = label) %>%
  add_global_graph_attrs(
    attr = "fontname",
    value = "Helvetica",
    attr_type = "edge") %>%
  add_global_graph_attrs(
    attr = "fontcolor",
    value = "gray50",
    attr_type = "edge") %>%
  add_global_graph_attrs(
    attr = "fontsize",
    value = 10,
    attr_type = "edge")

# Display this graph's
# metagraph, or, the underlying
# graph model for a property graph
display_metagraph(property_graph)

## End(Not run)
```

`do_bfs` *Use the breadth-first search (bfs) algorithm*

Description

With a chosen or random node serving as the starting point, perform a breadth-first search of the whole graph and return the node ID values visited. The bfs algorithm differs from depth-first search (dfs) in that bfs will follow tree branches branches one level at a time until terminating at leaf node (dfs traverses branches as far as possible).

Usage

```
do_bfs(graph, node = NULL, direction = "all")
```

Arguments

`graph`	a graph object of class `dgr_graph` that is created using `create_graph`.
`node`	an optional node ID value to specify a single starting point for the bfs. If not provided, a random node from the graph will be chosen.
`direction`	using `all` (the default), the bfs will ignore edge direction while traversing through the graph. With `out` and `in`, traversals between adjacent nodes will respect the edge direction.

Value

a vector containing node ID values for nodes visited during the breadth-first search. The order of the node IDs corresponds to the order visited.

Examples

```
# Create a graph containing
# two balanced trees
graph <-
  create_graph() %>%
  add_balanced_tree(
    k = 2, h = 2) %>%
  add_balanced_tree(
    k = 3, h = 2)

# Perform a breadth-first
# search of the graph,
# beginning at the root node
# `1` (the default
# `direction = "all"` doesn't
# take edge direction into
# account)
graph %>%
  do_bfs(node = 1)

# If not specifying a
# starting node, the function
# will begin the search from
# a random node
graph %>%
  do_bfs()

# It's also possible to
# perform bfs while taking
# into account edge direction;
# using `direction = "in"`
# causes the bfs routine to
# visit nodes along inward edges
graph %>%
  do_bfs(
    node = 1,
    direction = "in")

# Using `direction = "out"`
# results in the bfs moving
# along solely outward edges
graph %>%
  do_bfs(
    node = 1,
    direction = "out")
```

`do_dfs` *Use the depth-first search (dfs) algorithm*

Description

With a chosen or random node serving as the starting point, perform a depth-first search of the whole graph and return the node ID values visited. The dfs algorithm differs from breadth-first search (bfs) in that dfs will follow tree branches as far as possible until terminating at leaf node (bfs traverses branches one level at a time).

Usage

```
do_dfs(graph, node = NULL, direction = "all")
```

Arguments

graph	a graph object of class dgr_graph that is created using create_graph.
node	an optional node ID value to specify a single starting point for the dfs. If not provided, a random node from the graph will be chosen.
direction	using all (the default), the bfs will ignore edge direction while traversing through the graph. With out and in, traversals between adjacent nodes will respect the edge direction.

Value

a vector containing node ID values for nodes visited during the depth-first search. The order of the node IDs corresponds to the order visited.

Examples

```
# Create a graph containing
# two balanced trees
graph <-
  create_graph() %>%
  add_balanced_tree(
    k = 2, h = 2) %>%
  add_balanced_tree(
  k = 3, h = 2)

# Perform a depth-first
# search of the graph,
# beginning at the root
# node `1` (the default
# `direction = "all"`
# doesn't take edge
# direction into account)
graph %>%
  do_dfs(node = 1)

# If not specifying a
# starting node, the function
# will begin the search
# from a random node
graph %>%
```

```
  do_dfs()

# It's also possible to
# perform dfs while taking
# into account edge direction;
# using `direction = "in"`
# causes the dfs routine to
# visit nodes along inward edges
graph %>%
  do_dfs(
    node = 1,
    direction = "in")

# Using `direction = "out"`
# results in the dfs moving
# along solely outward edges
graph %>%
  do_dfs(
    node = 1,
    direction = "out")
```

---

drop_edge_attrs *Drop an edge attribute column*

Description

Within a graph's internal edge data frame (edf), remove an existing edge attribute.

Usage

```
drop_edge_attrs(graph, edge_attr)
```

Arguments

graph	a graph object of class dgr_graph.
edge_attr	the name of the edge attribute column to drop.

Value

a graph object of class dgr_graph.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 5,
```

```
    m = 6,
    set_seed = 23) %>%
  set_edge_attrs(
    edge_attr = value,
    values = 3) %>%
  mutate_edge_attrs(
    penwidth = value * 2)

# Get the graph's internal
# edf to show which edge
# attributes are available
graph %>%
  get_edge_df()

# Drop the `value` edge
# attribute
graph <-
  graph %>%
  drop_edge_attrs(
    edge_attr = value)

# Get the graph's internal
# edf to show that the edge
# attribute `value` had been
# removed
graph %>%
  get_edge_df()
```

`drop_node_attrs` *Drop a node attribute column*

Description

Within a graph's internal ndf, remove an existing node attribute.

Usage

```
drop_node_attrs(graph, node_attr)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`node_attr`	the name of the node attribute column to drop.

Value

a graph object of class `dgr_graph`.

Examples

```
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 5,
    m = 10,
    set_seed = 23) %>%
 set_node_attrs(
    node_attr = value,
    values = rnorm(
      n = count_nodes(.),
      mean = 5,
      sd = 1) %>% round(1))

# Get the graph's internal
# ndf to show which node
# attributes are available
graph %>%
  get_node_df()

# Drop the `value` node
# attribute
graph <-
  graph %>%
  drop_node_attrs(
    node_attr = value)

# Get the graph's internal
# ndf to show that the node
# attribute `value` had been
# removed
graph %>%
  get_node_df()
```

---

edge_aes *Insert edge aesthetic attributes during edge creation*

Description

This helper function should be invoked to provide values for the namesake edge_aes argument, which is present in any function where edges are created.

Usage

```
edge_aes(style = NULL, penwidth = NULL, color = NULL, arrowsize = NULL,
  arrowhead = NULL, arrowtail = NULL, fontname = NULL, fontsize = NULL,
  fontcolor = NULL, len = NULL, tooltip = NULL, URL = NULL,
  label = NULL, labelfontname = NULL, labelfontsize = NULL,
  labelfontcolor = NULL, labeltooltip = NULL, labelURL = NULL,
```

```
edgetooltip = NULL, edgeURL = NULL, dir = NULL, headtooltip = NULL,
headURL = NULL, headclip = NULL, headlabel = NULL, headport = NULL,
tailtooltip = NULL, tailURL = NULL, tailclip = NULL, taillabel = NULL,
tailport = NULL, decorate = NULL)
```

Arguments

style	the edge line style. The style types that can be used are solid, bold, dashed, dotted, tapered, and invisible.
penwidth	the thickness of the stroke line for the edge itself.
color	the color of the edge. Can be an X11 color or a hexadecimal color code.
arrowsize	a scaling factor for arrowheads. The default value is 1.0 and the minimum is 0.
arrowhead	the type of arrowhead to use. The style attribute can either any of these types: normal, vee, tee, dot, diamond, box, curve, icurve, inv, crow, or none.
arrowtail	the type of arrowtail to use. The style attribute can either any of these types: normal, vee, tee, dot, diamond, box, curve, icurve, inv, crow, or none.
fontname	the name of the system font that will be used for any edge text.
fontsize	the point size of the font used for any edge text.
fontcolor	the color used for any edge text. Can be an X11 color or a hexadecimal color code.
len	the preferred edge length for an edge, in inches. Default value is 1.0.
tooltip	text for a tooltip that appears when hovering over an edge. If text is not provided, then the default tooltip text will provide the edge definition (i.e., [id]->[id] or [id]--[id]).
URL	a URL to associate with an edge. Upon rendering the plot, clicking edges with any associated URLs will open the URL in the default browser.
label	the label text associated with the edge. This text will appear near the center of the edge.
labelfontname	the name of the system font that will be used for the headlabel and the taillabel label text. If not set, the fontname value will instead be used.
labelfontsize	the point size of the font used for the headlabel and the taillabel label text. If not set, the fontsize value will instead be used.
labelfontcolor	the color used for the label text of the headlabel and the taillabel label text. If not set, the fontcolor value will instead be used. Can be an X11 color or a hexadecimal color code.
labeltooltip	text for a tooltip that will appear when hovering over the main label of an edge (if label text provided in the label edge attribute). If text is not provided and an edge label is visible, then the default tooltip text will provide the edge definition (i.e., [id]->[id] or [id]--[id]).
labelURL	a URL to associate with edge label text. Upon rendering the plot, clicking edge labels with any associated URLs will open the URL in the default browser.
edgetooltip	this option provides a means to specify a tooltip with only the non-label parts of an edge. If this is defined, the value overrides any tooltip defined for the edge. This tooltip text is when hovering along the edge (even near the head or tail node) unless overridden by a headtooltip or tailtooltip value.

edgeURL — this option provides a means to specify a URL with only the non-label parts of an edge. If this is defined, the value overrides any URL defined for the edge. This URL is used along the edge (even near the head or tail node) unless overridden by a headURL or tailURL value.

dir — an optional direction type. Normally, for directed graphs, this is forward and needn't be set. For undirected graphs, this would be none and again no explicit setting is required. However, one can also use the back or both options. The back option draws an arrowhead in the reverse direction of an edge. The both option draws two arrowheads. When using any of these options in such an explicit manner, the head... and tail... edge attributes allow control over aesthetic edge attributes in either side of the edge.

headtooltip — this option provides a means to specify a tooltip that can be displayed by hovering over the part of an edge that is adjacent to incoming node (see the tooltip argument for further details).

headURL — this option provides a means to specify a URL that can be accessed by clicking the part of an edge that is adjacent to incoming node (see the URL argument for further details).

headclip — if TRUE (the default behavior), then the head of the affected edge is clipped to the node boundary. Using FALSE places the head of the outgoing edge at the center of its node.

headlabel — this option provides a means to display a label near the part of an edge that is adjacent to incoming node (see the label argument for further details).

headport — allows one to specify which compass position on the incoming node the head of the edge will alight. Options are n, ne, e, se, s, sw, w, and nw.

tailtooltip — this option provides a means to specify a tooltip that can be displayed by hovering over the part of an edge that is adjacent to outgoing node (see the tooltip argument for further details).

tailURL — this option provides a means to specify a URL that can be accessed by clicking the part of an edge that is adjacent to outgoing node (see the URL argument for further details).

tailclip — if TRUE (the default behavior), then the tail of the affected edge is clipped to the node boundary. Using FALSE places the tail of the outgoing edge at the center of its node.

taillabel — this option provides a means to display a label near the part of an edge that is adjacent to outgoing node (see the label argument for further details).

tailport — allows one to specify which compass position on the outgoing node the tail of the edge will be emitted from. Options are n, ne, e, se, s, sw, w, and nw.

decorate — if TRUE then attach any edge label to the edge line via a 2-segment polyline, underlining the label text and partially overlapping the edge line.

Examples

```
# Create a new graph and add
# a path with several edge
# aesthetic attributes
```

```
graph <-
  create_graph() %>%
  add_path(
    n = 3,
    type = "path",
    edge_aes = edge_aes(
      style = "dot",
      color = c("red", "blue")))

# View the graph's internal
# node data frame; the node
# aesthetic attributes have
# been inserted
graph %>%
  get_edge_df()
```

---

edge_data *Insert edge data attributes during edge creation*

Description

This helper function should be invoked to provide values for the namesake `edge_data` argument, which is present in any function where edges are created.

Usage

```
edge_data(...)
```

Arguments

`...`	edge data attributes provided as one or more named vectors.

Examples

```
## Not run:
# Create a new graph and add
# a path with several edge
# data attributes
graph <-
  create_graph() %>%
  add_path(
    n = 3,
    type = "path",
    edge_data = edge_data(
      hour = 5,
      index = c(1, 2)))

# View the graph's internal
# edge data frame; the edge
# data attributes have
```

```
# been inserted
graph %>%
  get_edge_df()

## End(Not run)
```

`edge_list_1` *Edge list - Version 1.*

Description

A very simple, 2-column data frame that can be used to generate graph edges.

Usage

```
edge_list_1
```

Format

A data frame with 19 rows and 2 variables:

from integer values that state the node ID values where an edge starts

to integer values that state the node ID values where an edge terminates

---

`edge_list_2` *Edge list - Version 2.*

Description

A simple, 5-column data frame that can be used to generate graph edges.

Usage

```
edge_list_2
```

Format

A data frame with 19 rows and 5 variables:

from integer values that state the node ID values where an edge starts

to integer values that state the node ID values where an edge terminates

rel a grouping variable of either a, b, or c

value_1 a randomized set of numeric values between 0 and 10

value_2 a randomized set of numeric values between 0 and 10

export_csv *Export a graph to CSV files*

Description

Export a graph to CSV files.

Usage

```
export_csv(graph, ndf_name = "nodes.csv", edf_name = "edges.csv",
  output_path = getwd(), colnames_type = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
ndf_name	the name to provide to the CSV file containing node information. By default this CSV will be called nodes.csv.
edf_name	the name to provide to the CSV file containing edge information. By default this CSV will be called edges.csv.
output_path	the path to which the CSV files will be placed. By default, this is the current working directory.
colnames_type	provides options to modify CSV column names to allow for easier import into other graph systems. The neo4j option modifies column names to allow for direct import of CSVs into Neo4J with the LOAD CSV clause. The graphframes option modifies column names to match those required by the Spark GraphFrames package.

Examples

```
# Create a node data frame (ndf)
ndf <-
  create_node_df(
    n = 4,
    type = c("a", "a", "z", "z"),
    label = TRUE,
    value = c(3.5, 2.6, 9.4, 2.7))

# Create an edge data frame (edf)
edf <-
  create_edge_df(
    from = c(1, 2, 3),
    to = c(4, 3, 1),
    rel = c("rel_a", "rel_z", "rel_a"))

# Create a graph with the ndf and edf
graph <-
  create_graph(
```

```
    nodes_df = ndf,
    edges_df = edf)

# Create separate `nodes.csv` and `edges.csv`
# files in the working directory
graph %>%
  export_csv()
```

---

export_graph *Export a graph to various file formats*

Description

Export a graph to a variety of file formats, including image formats such as PNG, PDF, SVG, and PostScript, and graph file formats such as GEXF.

Usage

```
export_graph(graph, file_name = NULL, file_type = NULL, title = NULL,
  width = NULL, height = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
file_name	the name of the exported file (including it's extension).
file_type	the type of file to be exported. Options for graph files are: png, pdf, svg, and ps. Options for graph file formats are: gexf.
title	an optional title for the output graph.
width	output width in pixels or NULL for default. Only useful for export to image file formats png, pdf, svg, and ps.
height	output height in pixels or NULL for default. Only useful for export to image file formats png, pdf, svg, and ps.

Examples

```
## Not run:
# Create a simple graph
graph <-
  create_graph() %>%
    add_path(
      n = 5,
      edge_aes = edge_aes(
        arrowhead = c(
          "normal", "vee",
          "tee", "dot"),
        color = c(
        "red", "blue",
```

```
        "orange", "purple")))

# Create a PDF file for
# the graph (`graph.pdf`)
graph %>%
  export_graph(
    file_name = "graph.pdf",
    title = "Simple Graph")

# Create a PNG file for
# the graph (`mypng.png`)
graph %>%
  export_graph(
    file_name = "mypng.png",
    file_type = "PNG")

## End(Not run)
```

`filter_graph_series` *Subset a graph series object*

Description

Subsetting a graph series by the graphs' index positions in the graph series or through selection via graphs' date-time attributes.

Usage

```
filter_graph_series(graph_series, by = "number", values, tz = NULL)
```

Arguments

`graph_series`	a graph series object of type `dgr_graph_1D`.
`by`	either `number`, which allows for subsetting of the graph series by graph indices, or `time` which for graph series objects of type `temporal` allows for a subsetting of graphs by a date-time or time range.
`values`	where the subsetting of the graph series by to occur via graph indices (where `by = number`), provide a vector of those indices; when subsetting by time (where `by = time`), a range of times can be provided as a vector.
`tz`	the time zone (`tz`) corresponding to dates or date-time string provided in `values` (if `by = "date"`).

Value

a graph series object of type `dgr_graph_1D`.

Examples

```
# Create three graphs
graph_time_1 <-
  create_graph(
    graph_name = "graph_with_time_1") %>%
  set_graph_time(
    time = "2015-03-25 03:00",
    tz = "GMT")

graph_time_2 <-
  create_graph(
    graph_name = "graph_with_time_2") %>%
  set_graph_time(
    time = "2015-03-26 03:00",
    tz = "GMT")

graph_time_3 <-
  create_graph(
    graph_name = "graph_with_time_3") %>%
  set_graph_time(
    time = "2015-03-27 15:00",
    tz = "GMT")

# Create an empty graph series and add
# the graphs
series_temporal <-
  create_graph_series(
    series_type = "temporal") %>%
  add_graph_to_graph_series(
    graph = graph_time_1) %>%
  add_graph_to_graph_series(
    graph = graph_time_2) %>%
  add_graph_to_graph_series(
    graph = graph_time_3)

# Subset graph series by sequence
series_sequence_subset <-
  filter_graph_series(
    graph_series = series_temporal,
    by = "number",
    values = 2)

# Get a count of graphs in
# the series
series_sequence_subset %>%
  count_graphs_in_graph_series()

# Subset graph series by date-time
series_time_subset <-
  filter_graph_series(
    graph_series = series_temporal,
    by = "time",
```

```
    values = c("2015-03-25 12:00",
               "2015-03-26 12:00"),
    tz = "GMT")

# Get a count of graphs in
# the series
series_time_subset %>%
  count_graphs_in_graph_series()
```

`from_adj_matrix` *Create a graph using an adjacency matrix*

Description

Using an adjacency matrix object, generate a graph of class `dgr_graph`.

Usage

```
from_adj_matrix(x, mode = "undirected", weighted = FALSE, use_diag = TRUE,
  graph_name = NULL, write_backups = FALSE)
```

Arguments

`x`	a square `matrix` object serving as the adjacency matrix.
`mode`	the method in which to interpret the input adjacency matrix. Options include: `undirected`, `directed`, `upper`, `lower`, `max`, `min`, and `plus`.
`weighted`	whether to create a weighted graph from the adjacency matrix.
`use_diag`	whether to use the diagonal of the adjacency matrix in calculations. If `TRUE` then the diagonal values will be included as is. If `FALSE` then the diagonal values will be replaced with zero values before inclusion in any calculations.
`graph_name`	an optional string for labeling the graph object.
`write_backups`	an option to write incremental backups of changing graph states to disk. If `TRUE`, a subdirectory of the working directory will be used to store `RDS` files. The default value is `FALSE` so one has to opt in to use this functionality.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create an adjacency matrix
adj_matrix <-
  sample(0:1, 100,
         replace = TRUE,
         prob = c(0.9,0.1)) %>%
  matrix(nc = 10)
```

```
# Create a graph from the adjacency matrix
graph <-
  from_adj_matrix(adj_matrix)
```

---

from_igraph *Convert an igraph graph to a DiagrammeR one*

Description

Convert an igraph graph to a DiagrammeR graph object.

Usage

```
from_igraph(igraph, graph_name = NULL, write_backups = FALSE)
```

Arguments

igraph	an igraph graph object.
graph_name	an optional string for labeling the graph object.
write_backups	an option to write incremental backups of changing graph states to disk. If TRUE, a subdirectory of the working directory will be used to store RDS files. The default value is FALSE so one has to opt in to use this functionality.

Value

a graph object of class dgr_graph.

Examples

```
# Create a DiagrammeR graph object
dgr_graph_orig <-
  create_graph() %>%
  add_gnm_graph(
    n = 36,
    m = 50,
    set_seed = 23)

# Convert the DiagrammeR
# graph to an igraph object
ig_graph <-
  dgr_graph_orig %>%
  to_igraph()

# Convert the igraph graph
# back to a DiagrammeR graph
dgr_graph_new <-
  ig_graph %>%
  from_igraph()
```

```
# Get some graph information
(dgr_graph_new %>%
  get_graph_info())[, 1:6]
```

`fully_connect_nodes_ws` *Fully connect all nodes in a selection of nodes*

Description

With a selection of nodes in a graph, add any remaining edges required to fully connect this group of edges to each other.

Usage

```
fully_connect_nodes_ws(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create an empty graph and
# then add a path of 3 nodes
# and two isolated nodes
graph <-
  create_graph() %>%
  add_path(n = 3) %>%
  add_n_nodes(n = 2)

# Select a node in the path
# of nodes (node `3`) and
# the two isolated nodes (`4`
# and `5`); then, and fully
# connect these nodes together
graph <-
  graph %>%
  select_nodes_by_id(
    nodes = 3:5) %>%
  fully_connect_nodes_ws()

# Get the graph's edge data frame
graph %>%
  get_edge_df()
```

```
# Create an undirected, empty
# graph; add a path of 3 nodes
# and two isolated nodes
graph <-
  create_graph(
    directed = FALSE) %>%
  add_path(n = 3) %>%
  add_n_nodes(n = 2)

# Select a node in the path
# of nodes (node `3`) and
# the two isolated nodes (`4`
# and `5`); then, and fully
# connect these nodes together
graph <-
  graph %>%
  select_nodes_by_id(
    nodes = 3:5) %>%
  fully_connect_nodes_ws()

# Get the graph's edge data
# frame; in the undirected
# case, reverse edges aren't
# added
graph %>%
  get_edge_df()
```

`fully_disconnect_nodes_ws` *Fully disconnect all nodes in a selection of nodes*

Description

With a selection of nodes in a graph, remove any edges to or from those nodes.

Usage

```
fully_disconnect_nodes_ws(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create an empty graph and
# add a path of 6 nodes
graph <-
  create_graph() %>%
  add_path(n = 6)

# Select nodes `3` and `4`
# and fully disconnect them
# from the graph
graph <-
  graph %>%
  select_nodes_by_id(
    nodes = 3:4) %>%
  fully_disconnect_nodes_ws()

# Get the graph's edge data frame
graph %>%
  get_edge_df()
```

`generate_dot` *Generate DOT code using a graph object*

Description

Generates Graphviz DOT code as an R character object using DiagrammeR graph object.

Usage

```
generate_dot(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a character vector of length 1 containing Graphviz DOT code.

get_adhesion *Get graph adhesion*

Description

Get the adhesion of a graph, which is the minimum number of edges needed to remove to obtain a graph which is not strongly connected. This is the same as the edge connectivity of the graph.

Usage

```
get_adhesion(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a single numeric value representing the minimum number of edges to remove.

Examples

```
# Create a cycle graph
graph <-
  create_graph() %>%
  add_cycle(n = 5)

# Determine the graph's adhesion
graph %>%
  get_adhesion()

# Create a full graph and then
# get the adhesion for that
create_graph() %>%
  add_full_graph(n = 8) %>%
  get_adhesion()
```

get_agg_degree_in *Get an aggregate value from the indegree of nodes*

Description

Get a single, aggregate value from the indegree values for all nodes in a graph, or, a subset of graph nodes.

Usage

```
get_agg_degree_in(graph, agg, conditions = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
agg	the aggregation function to use for summarizing indegree values from graph nodes. The following aggregation functions can be used: sum, min, max, mean, or median.
conditions	an option to use filtering conditions for the nodes to consider.

Value

a vector with an aggregate indegree value.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 20,
    m = 35,
    set_seed = 23) %>%
  set_node_attrs(
    node_attr = value,
    values = rnorm(
      n = count_nodes(.),
      mean = 5,
      sd = 1) %>% round(1))

# Get the mean indegree value
# from all nodes in the graph
graph %>%
  get_agg_degree_in(
    agg = "mean")

# Other aggregation functions
# can be used (`min`, `max`,
# `median`, `sum`); let's get
# the median in this example
graph %>%
  get_agg_degree_in(
    agg = "median")

# The aggregation of indegree
# can occur for a subset of the
# graph nodes and this is made
# possible by specifying
# `conditions` for the nodes
graph %>%
  get_agg_degree_in(
    agg = "mean",
    conditions = value > 5.0)
```

get_agg_degree_out *Get an aggregate value from the outdegree of nodes*

Description

Get a single, aggregate value from the outdegree values for all nodes in a graph, or, a subset of graph nodes.

Usage

```
get_agg_degree_out(graph, agg, conditions = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
agg	the aggregation function to use for summarizing outdegree values from graph nodes. The following aggregation functions can be used: sum, min, max, mean, or median.
conditions	an option to use filtering conditions for the nodes to consider.

Value

a vector with an aggregate outdegree value.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 20,
    m = 35,
    set_seed = 23) %>%
  set_node_attrs(
    node_attr = value,
    values = rnorm(
      n = count_nodes(.),
      mean = 5,
      sd = 1) %>% round(1))

# Get the mean outdegree value from all
# nodes in the graph
graph %>%
  get_agg_degree_out(
    agg = "mean")

# Other aggregation functions can be used
# (`min`, `max`, `median`, `sum`); let's
```

```
# get the median in this example
graph %>%
  get_agg_degree_out(
    agg = "median")

# The aggregation of outdegree can occur
# for a subset of the graph nodes and this
# is made possible by specifying `conditions`
# for the nodes
graph %>%
  get_agg_degree_out(
    agg = "mean",
    conditions = value < 5.0)
```

`get_agg_degree_total` *Get an aggregate value from the total degree of nodes*

Description

Get a single, aggregate value from the total degree values for all nodes in a graph, or, a subset of graph nodes.

Usage

```
get_agg_degree_total(graph, agg, conditions = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`agg`	the aggregation function to use for summarizing total degree values from graph nodes. The following aggregation functions can be used: `sum`, `min`, `max`, `mean`, or `median`.
`conditions`	an option to use filtering conditions for the nodes to consider.

Value

a vector with an aggregate total degree value.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 20,
    m = 35,
    set_seed = 23) %>%
  set_node_attrs(
```

```
    node_attr = value,
    values = rnorm(
      n = count_nodes(.),
      mean = 5,
      sd = 1) %>% round(1))

# Get the mean total degree
# value from all nodes in
# the graph
graph %>%
  get_agg_degree_total(
    agg = "mean")

# Other aggregation functions
# can be used (`min`, `max`,
# `median`, `sum`); let's get
# the median in this example
graph %>%
  get_agg_degree_total(
    agg = "median")

# The aggregation of total
# degree can occur for a
# subset of the graph nodes
# and this is made possible
# by specifying `conditions`
# for the nodes
graph %>%
  get_agg_degree_total(
    agg = "mean",
    conditions = value < 5.0)
```

`get_all_connected_nodes` *Get all nodes connected to a specified node*

Description

With a single node serving as the starting point get all nodes connected (i.e., reachable with a traversable path) to that node.

Usage

```
get_all_connected_nodes(graph, node)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`node`	a single-length vector containing a node ID value.

Value

a vector of node ID values.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function; it
# has an unconnected node (`6`)
graph_1 <-
  create_graph() %>%
  add_gnm_graph(
    n = 20,
    m = 32,
    set_seed = 23)

# There won't be any connected
# nodes to `6` so when specifying
# this node with `get_all_connected_nodes()`
# we get NA back
graph_1 %>%
  get_all_connected_nodes(
    node = 6)

# Any other node in `graph_1` will
# provide a vector of all the nodes
# other than `6`
graph_1 %>%
  get_all_connected_nodes(
    node = 1)

# The following graph has two
# clusters of nodes (i.e., the
# graph has two connected components)
graph_2 <-
  create_graph() %>%
  add_path(n = 6) %>%
  add_path(n = 4)

# In `graph_2`, node `1` is in
# the larger of the two
# connected components
graph_2 %>%
  get_all_connected_nodes(
    node = 1)

# Also in `graph_2`, node `8`
# is in the smaller of the two
# connected components
graph_2 %>%
  get_all_connected_nodes(
    node = 8)
```

get_alpha_centrality *Get the alpha centrality for all nodes*

Description

Get the alpha centrality values for all nodes in the graph.

Usage

```
get_alpha_centrality(graph, alpha = 1, exo = 1, weights_attr = NULL,
  tol = 1e-07)
```

Arguments

graph	a graph object of class dgr_graph.
alpha	the parameter that specifies the relative importance of endogenous versus exogenous factors in the determination of centrality.
exo	the exogenous factors, in most cases this is either a constant (which applies the same factor to every node), or a vector giving the factor for every node.
weights_attr	an optional name of the edge attribute to use in the adjacency matrix. If NULL then, if it exists, the weight edge attribute of the graph will be used. Failing that, the standard adjacency matrix will be used in calculations.
tol	the tolerance for near-singularities during matrix inversion. Default value is set to 1e-7.

Value

a data frame with alpha centrality scores for each of the nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 10,
    m = 12,
    set_seed = 23)

# Get the alpha centrality scores
# for all nodes
graph %>%
  get_alpha_centrality()

# Add the alpha centrality
# scores to the graph as a node
```

```
# attribute
graph <-
  graph %>%
  join_node_attrs(
    df = get_alpha_centrality(.))

# Display the graph's node
# data frame
graph %>%
  get_node_df()
```

`get_articulation_points` *Get articulation points*

Description

Get the nodes in the graph that are identified as articulation points.

Usage

```
get_articulation_points(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a vector of node IDs.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 10,
    m = 12,
    set_seed = 23) %>%
  set_node_attrs(
    node_attr = shape,
    values = "square")

# Get the articulation points
# in the graph (i.e., those
# nodes that if any were to be
# removed, the graph would
# become disconnected)
```

```
graph %>%
  get_articulation_points()

# For the articulation points,
# change the node shape to
# a `circle`
graph <-
  graph %>%
  select_nodes_by_id(
    nodes = get_articulation_points(.)) %>%
  set_node_attrs_ws(
    node_attr = shape,
    value = "circle")
```

`get_attr_dfs` *Get data frames bound to node attributes*

Description

From a graph object of class `dgr_graph`, get one or more data frames already bound as node and/or edge attribute values given graph node and/or edges.

Usage

```
get_attr_dfs(graph, node_id = NULL, edge_id = NULL,
  return_format = "single_tbl")
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`node_id`	a vector of node ID values in which data frames are bound as node attrs.
`edge_id`	a vector of edge ID values in which data frames are bound as edge attrs.
`return_format`	the format in which to return the results of several data frames. These can either be: (1) `single_tbl` (a tibble object resulting from a 'bind_rows' operation of multiple data frames), and (2) `single_df` (a single data frame which all of the data frame data).

Value

either a tibble or a data frame.

Examples

```
# Create a node data frame (ndf)
ndf <-
  create_node_df(
    n = 4,
    type = "basic",
```

```
    label = TRUE,
    value = c(3.5, 2.6, 9.4, 2.7))

# Create an edge data frame (edf)
edf <-
  create_edge_df(
    from = c(1, 2, 3),
    to = c(4, 3, 1),
    rel = "leading_to")

# Create a graph
graph <-
  create_graph(
    nodes_df = ndf,
    edges_df = edf)

# Create 3 simple data frames to add as
# attributes to nodes/edges
df_1 <-
  data.frame(
    a = c("one", "two"),
    b = c(1, 2),
    stringsAsFactors = FALSE)

df_2 <-
  data.frame(
    a = c("three", "four"),
    b = c(3, 4),
    stringsAsFactors = FALSE)

df_for_edges <-
  data.frame(
    c = c("five", "six"),
    d = c(5, 6),
    stringsAsFactors = FALSE)

# Bind data frames as node attributes
# for nodes `1` and `4`; bind a data
# frame as an edge attribute as well
graph <-
  graph %>%
  set_df_as_node_attr(
    node = 1,
    df = df_1) %>%
  set_df_as_node_attr(
    node = 4,
    df = df_2) %>%
  set_df_as_edge_attr(
    edge = 1,
    df = df_for_edges)

# Get a single tibble by specifying the
# nodes from which there are data frames
```

```
# bound as node attributes
get_attr_dfs(
  graph,
  node_id = c(1, 4))

# You can also get data frames that are
# associated with edges by using the
# same function
get_attr_dfs(
  graph,
  edge_id = 1)

# It's also possible to collect data frames
# associated with both nodes and edges
get_attr_dfs(
  graph,
  node_id = 4,
  edge_id = 1)

# If a data frame is desired instead,
# set `return_format = "single_df"`
get_attr_dfs(
  graph,
  edge_id = 1,
  return_format = "single_df")
```

`get_authority_centrality`

Get the authority scores for all nodes

Description

Get the Kleinberg authority centrality scores for all nodes in the graph.

Usage

```
get_authority_centrality(graph, weights_attr = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`weights_attr`	an optional name of the edge attribute to use in the adjacency matrix. If `NULL` then, if it exists, the `weight` edge attribute of the graph will be used.

Value

a data frame with authority scores for each of the nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Get the authority centrality scores
# for all nodes in the graph
graph %>%
  get_authority_centrality()

# Add the authority centrality
# scores to the graph as a node
# attribute
graph <-
  graph %>%
  join_node_attrs(
    df = get_authority_centrality(.))

# Display the graph's node data frame
graph %>%
  get_node_df()
```

---

get_betweenness *Get betweenness centrality scores*

Description

Get the betweenness centrality scores for all nodes in a graph.

Usage

```
get_betweenness(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a data frame with betweenness scores for each of the nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 10,
    m = 12,
    set_seed = 23)

# Get the betweenness scores
# for nodes in the graph
graph %>%
  get_betweenness()

# Add the betweenness
# values to the graph
# as a node attribute
graph <-
  graph %>%
  join_node_attrs(
    df = get_betweenness(.))

# Display the graph's node
# data frame
graph %>%
  get_node_df()
```

---

get_bridging *Get bridging scores*

Description

Get the bridging scores (based on Valente's Bridging vertex measure) for all nodes in a graph.

Usage

```
get_bridging(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a data frame with bridging scores for each of the nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 10,
    m = 12,
    set_seed = 23)

# Get the bridging scores for nodes
# in the graph
graph %>%
  get_bridging()

# Add the bridging scores to
# the graph as a node attribute
graph <-
  graph %>%
  join_node_attrs(
    df = get_bridging(.))

# Display the graph's node data frame
graph %>%
  get_node_df()
```

---

get_cache *Get a cached vector from a graph object*

Description

Get the vector cached in a graph object of class dgr_graph.

Usage

```
get_cache(graph, name = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
name	the name of the object to extract from the cache. If none supplied, the most recent object added to the cache will be returned.

Value

a vector.

Examples

```
# Set a seed
set.seed(23)

# Create a graph with 5 nodes and 5 edges
graph <-
  create_graph() %>%
  add_n_nodes(n = 5) %>%
  set_node_attrs(
    node_attr = value,
    values = rnorm(
      n = count_nodes(.),
      mean = 8,
      sd = 2)) %>%
  add_edges_w_string(
    edges = "1->2 1->3 2->4 2->5 3->2")

# Cache all values from the node attribute `value`
# as a numeric vector
graph <-
  graph %>%
  set_cache(
    name = "value",
    to_cache = get_node_attrs(
      graph = .,
      node_attr = value))

# Return the cached vector
graph %>%
  get_cache()
```

---

get_closeness *Get closeness centrality values*

Description

Get the closeness centrality values for all nodes in a graph.

Usage

```
get_closeness(graph, direction = "all")
```

Arguments

graph	a graph object of class dgr_graph.
direction	using all (the default), the search will ignore edge direction while traversing through the graph. With out, measurements of paths will be from a node whereas with in, measurements of paths will be to a node.

Value

a data frame with closeness values for each of the nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 10,
    m = 12,
    set_seed = 23)

# Get closeness values for all nodes
# in the graph
graph %>%
  get_closeness()

# Add the closeness values to
# the graph as a node attribute
graph <-
  graph %>%
  join_node_attrs(
    df = get_closeness(.))

# Display the graph's node data frame
graph %>%
  get_node_df()
```

---

get_closeness_vitality

Get closeness vitality

Description

Get the closeness vitality values for all nodes in the graph.

Usage

```
get_closeness_vitality(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a data frame with closeness vitality values for each of the nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 10,
    m = 12,
    set_seed = 23)

# Get closeness vitality values
# for all nodes in the graph
graph %>%
  get_closeness_vitality()

# Add the closeness vitality
# values to the graph as a
# node attribute
graph <-
  graph %>%
  join_node_attrs(
    df = get_closeness_vitality(.))

# Display the graph's
# node data frame
graph %>%
  get_node_df()
```

---

get_cmty_edge_btwns *Get community membership by edge betweenness*

Description

Using edge betweenness, obtain the group membership values for each of the nodes in the graph.

Usage

```
get_cmty_edge_btwns(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a data frame with group membership assignments for each of the nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Get the group membership
# values for all nodes in the
# graph through calculation of
# the leading non-negative
# eigenvector of the modularity
# matrix of the graph
graph %>%
  get_cmty_edge_btwns()

# Add the group membership
# values to the graph
# as a node attribute
graph <-
  graph %>%
  join_node_attrs(
     df = get_cmty_edge_btwns(.))

# Display the graph's
# node data frame
graph %>%
  get_node_df()
```

---

get_cmty_fast_greedy *Get community membership by modularity optimization*

Description

Through the use of greedy optimization of a modularity score, obtain the group membership values for each of the nodes in the graph. Note that this method only works on graphs without multiple edges.

Usage

```
get_cmty_fast_greedy(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a data frame with group membership assignments for each of the nodes.

Examples

```
# Create a graph with a
# balanced tree
graph <-
  create_graph() %>%
  add_balanced_tree(
    k = 2,
    h = 2)

# Get the group membership
# values for all nodes in
# the graph through the greedy
# optimization of modularity
# algorithm
graph %>%
  get_cmty_fast_greedy()

# Add the group membership
# values to the graph as a
# node attribute
graph <-
  graph %>%
  join_node_attrs(
    df = get_cmty_fast_greedy(.))

# Display the graph's
# node data frame
graph %>%
  get_node_df()
```

---

get_cmty_louvain *Get community membership by Louvain optimization*

Description

Through the use of multi-level optimization of a modularity score, obtain the group membership values for each of the nodes in the graph.

Usage

```
get_cmty_louvain(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a data frame with group membership assignments for each of the nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Get the group membership values
# for all nodes in the graph
# through the multi-level
# optimization of modularity
# algorithm
graph %>%
  get_cmty_louvain()

# Add the group membership
# values to the graph as a
# node attribute
graph <-
  graph %>%
  join_node_attrs(
    df = get_cmty_louvain(.))

# Display the graph's
# node data frame
graph %>%
  get_node_df()
```

---

get_cmty_l_eigenvec *Get community membership by leading eigenvector*

Description

Through the calculation of the leading non-negative eigenvector of the modularity matrix of the graph, obtain the group membership values for each of the nodes in the graph.

Usage

```
get_cmty_l_eigenvec(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a data frame with group membership assignments for each of the nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Get the group membership
# values for all nodes in the
# graph through calculation of
# the leading non-negative
# eigenvector of the modularity
# matrix of the graph
graph %>%
  get_cmty_l_eigenvec()

# Add the group membership
# values to the graph as a node
# attribute
graph <-
  graph %>%
  join_node_attrs(
    df = get_cmty_l_eigenvec(.))

# Display the graph's node data frame
graph %>%
  get_node_df()
```

---

get_cmty_walktrap *Get community membership using the Walktrap method*

Description

With the Walktrap community finding algorithm, obtain the group membership values for each of the nodes in the graph.

Usage

```
get_cmty_walktrap(graph, steps = 4)
```

Arguments

graph	a graph object of class dgr_graph.
steps	the number of steps to take for each of the random walks.

Value

a data frame with group membership assignments for each of the nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Get the group membership
# values for all nodes in the
# graph through the Walktrap
# community finding algorithm
graph %>%
  get_cmty_walktrap()

# Add the group membership
# values to the graph as a
# node attribute
graph <-
  graph %>%
  join_node_attrs(
    df = get_cmty_walktrap(.))

# Display the graph's
# node data frame
graph %>%
  get_node_df()
```

---

get_common_nbrs *Get all common neighbors between two or more nodes*

Description

With two or more nodes, get the set of common neighboring nodes.

Usage

```
get_common_nbrs(graph, nodes)
```

Arguments

graph	a graph object of class dgr_graph.
nodes	a vector of node ID values of length at least 2.

Value

a vector of node ID values.

Examples

```
# Create a directed graph with 5 nodes
graph <-
  create_graph() %>%
  add_path(n = 5)

# Find all common neighbor nodes
# for nodes `1` and `2` (there are no
# common neighbors amongst them)
graph %>%
  get_common_nbrs(
    nodes = c(1, 2))

# Find all common neighbor nodes for
# nodes `1` and `3`
graph %>%
  get_common_nbrs(
    nodes = c(1, 3))
```

---

get_constraint *Get constraint scores for one or more graph nodes*

Description

Get the constraint scores (based on Burt's Constraint Index) for one or more nodes in a graph.

Usage

```
get_constraint(graph, nodes = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
nodes	an optional vector of node IDs to consider for constraint scores. If not supplied, then constraint scores for all nodes in the graph will be calculated.

Value

a data frame with constraint scores for one or more graph nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Get the constaint scores for all
# nodes in the graph
graph %>%
  get_constraint()

# Get the constaint scores
# for nodes `5` and `7`
graph %>%
  get_constraint(
    nodes = c(5, 7))

# Add the constraint scores
# to the graph as a node
# attribute
graph <-
  graph %>%
  join_node_attrs(
    df = get_constraint(.))

# Display the graph's node data frame
graph %>%
  get_node_df()
```

---

get_coreness *Get coreness values for graph nodes*

Description

Get the coreness values for all nodes in a graph.

Usage

```
get_coreness(graph, direction = "all")
```

Arguments

graph	a graph object of class dgr_graph.
direction	using all (the default), the search will ignore edge direction while traversing through the graph. With out, measurements of paths will be from a node whereas with in, measurements of paths will be to a node.

Value

a data frame with coreness values for each of the nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Get coreness values for
# all nodes in the graph
graph %>%
  get_coreness()

# Add the coreness values
# to the graph as a node
# attribute
graph <-
  graph %>%
  join_node_attrs(
    df = get_coreness(.))

# Display the graph's node data frame
graph %>%
  get_node_df()
```

---

get_degree_distribution

Get total degree distribution data for a graph

Description

Get degree distribution data for a graph. Graph degree is represented as a frequency of total degree values over all nodes in the graph.

Usage

```
get_degree_distribution(graph, mode = "total")
```

Arguments

graph	a graph object of class dgr_graph.
mode	using total (the default), degree considered for each node will be the total degree. With in and out the degree used will be the in-degree and out-degree, respectively.

Value

a data frame with degree frequencies.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Get the total degree
# distribution for the graph
graph %>%
  get_degree_distribution(
    mode = "total")
```

---

get_degree_histogram *Get histogram data for a graph's degree frequency*

Description

Get histogram data for a graph's degree frequency. The bin width is set to 1 and zero-value degrees are omitted from the output.

Usage

```
get_degree_histogram(graph, mode = "total")
```

Arguments

graph	a graph object of class dgr_graph.
mode	using total (the default), degree considered for each node will be the total degree. With in and out the degree used will be the in-degree and out-degree, respectively.

Value

a data frame with degree counts.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Get degree histogram data for
# the graph (reporting total degree)
graph %>%
  get_degree_histogram(
    mode = "total")
```

---

get_degree_in *Get indegree values for all nodes*

Description

Get the indegree values for all nodes in a graph.

Usage

```
get_degree_in(graph, normalized = FALSE)
```

Arguments

graph	a graph object of class dgr_graph.
normalized	set as FALSE (the default), the indegree will be provided for each of the nodes (as a count of edges to each node). When set as TRUE, then the result for each node will be divided by the total number of nodes in the graph minus 1.

Value

a data frame with indegree values for each of the nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Get the indegree values for
# all nodes in the graph
graph %>%
  get_degree_in()

# Add the indegree values
# to the graph as a node
# attribute
graph <-
  graph %>%
  join_node_attrs(
    df = get_degree_in(.))

# Display the graph's
# node data frame
graph %>%
  get_node_df()
```

---

get_degree_out *Get outdegree values for all nodes*

Description

Get the outdegree values for all nodes in a graph.

Usage

```
get_degree_out(graph, normalized = FALSE)
```

Arguments

graph	a graph object of class dgr_graph.
normalized	set as FALSE (the default), the outdegree will be provided for each of the nodes (as a count of edges outgoing from each node). When set as TRUE, then the result for each node will be divided by the total number of nodes in the graph minus 1.

Value

a data frame with outdegree values for each of the nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Get the outdegree values
# for all nodes in the graph
graph %>%
  get_degree_out()

# Add the outdegree values
# to the graph as a node
# attribute
graph <-
  graph %>%
  join_node_attrs(
    df = get_degree_out(.))

# Display the graph's
# node data frame
graph %>%
  get_node_df()
```

---

get_degree_total *Get total degree values for all nodes*

Description

Get the total degree values for all nodes in a graph.

Usage

```
get_degree_total(graph, normalized = FALSE)
```

Arguments

graph	a graph object of class dgr_graph.
normalized	set as FALSE (the default), the total degree will be provided for each of the nodes (as a count of edges to and from each node). When set as TRUE, then the result for each node will be divided by the total number of nodes in the graph minus 1.

Value

a data frame with total degree values for each of the nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Get the total degree values
# for all nodes in the graph
graph %>%
  get_degree_total()

# Add the total degree values
# to the graph as a node
# attribute
graph <-
  graph %>%
  join_node_attrs(
    df = get_degree_total(.))

# Display the graph's
# node data frame
graph %>%
  get_node_df()
```

---

get_dice_similarity *Get Dice similarity coefficient scores*

Description

Get the Dice similarity coefficient scores for one or more nodes in a graph.

Usage

```
get_dice_similarity(graph, nodes = NULL, direction = "all", round_to = 3)
```

Arguments

graph	a graph object of class dgr_graph.

nodes	an optional vector of node IDs to consider for Dice similarity scores. If not supplied, then similarity scores will be provided for every pair of nodes in the graph.
direction	using `all` (the default), the function will ignore edge direction when determining scores for neighboring nodes. With `out` and `in`, edge direction for neighboring nodes will be considered.
round_to	the maximum number of decimal places to retain for the Dice similarity coefficient scores. The default value is `3`.

Value

a matrix with Dice similarity values for each pair of nodes considered.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Get the Dice similarity
# values for nodes `5`, `6`,
# and `7`
graph %>%
  get_dice_similarity(
    nodes = 5:7)
```

---

get_eccentricity *Get node eccentricities*

Description

Get a data frame with node eccentricity values.

Usage

```
get_eccentricity(graph, mode = "out")
```

Arguments

graph	a graph object of class dgr_graph.
mode	the mode with which the shortest paths to or from the given vertices should be calculated for directed graphs. If out (the default) then the shortest paths from the node, if in then only shortest paths to each node are considered. If all is used, then the corresponding undirected graph will be used and edge directions will be ignored. For undirected graphs, this argument is ignored.

Value

a data frame containing eccentricity values by node ID value.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Get the eccentricity values for
# all nodes in the graph
graph %>%
  get_eccentricity()
```

---

get_edges *Get node IDs associated with edges*

Description

Obtain a vector, data frame, or list of node IDs associated with edges in a graph object. An optional filter by edge attribute can limit the set of edges returned.

Usage

```
get_edges(graph, conditions = NULL, return_type = "vector",
  return_values = "id")
```

Arguments

graph	a graph object of class dgr_graph.
conditions	an option to use filtering conditions for the retrieval of edges.

return_type	using vector (the default), a vector of character objects representing the edges is provided. With list a list object will be provided that contains vectors of outgoing and incoming node IDs associated with edges. With df, a data frame containing outgoing and incoming node IDs associated with edges.
return_values	using id (the default) results in node ID values returned in the edge definitions. With label, the node labels will instead be used to define edges.

Value

a list, data frame, or a vector object, depending on the value given to return_type.

Examples

```
# Create a node data frame (ndf)
ndf <-
  create_node_df(
    n = 4,
    label = c("one", "two", "three", "four"),
    type = "letter",
    color = c("red", "green", "grey", "blue"),
    value = c(3.5, 2.6, 9.4, 2.7))

# Create an edge data frame (edf)
edf <-
  create_edge_df(
    from = c(1, 2, 3),
    to = c(4, 3, 1),
    rel = "leading_to",
    color = c("pink", "blue", "blue"),
    value = c(3.9, 2.5, 7.3))

# Create a graph
graph <-
  create_graph(
    nodes_df = ndf,
    edges_df = edf)

# Get all edges within a graph, returned as a list
graph %>%
  get_edges(
    return_type = "vector")

# Get all edges within a graph, returned as a
# data frame
graph %>%
  get_edges(
    return_type = "df")

# Get all edges returned as a list
graph %>%
  get_edges(
    return_type = "list")
```

```
# Get a vector of edges using
# a numeric comparison (i.e.,
# all edges with a `value`
# attribute greater than 3)
graph %>%
  get_edges(
    conditions = value > 3,
    return_type = "vector")

# Get a vector of edges using
# a matching condition
graph %>%
  get_edges(
    conditions = color == "pink",
    return_type = "vector")

# Use multiple conditions to
# return edges with the
# desired attribute values
graph %>%
  get_edges(
    conditions =
      color == "blue" &
      value > 3,
    return_type = "vector")

# Use `return_values = "label"`
# to return the labels of the
# connected nodes
graph %>%
  get_edges(
    conditions =
      color == "blue" &
      value > 3,
    return_type = "vector",
    return_values = "label")
```

`get_edge_attrs` *Get edge attribute values*

Description

From a graph object of class `dgr_graph`, get edge attribute values for one or more edges.

Usage

```
get_edge_attrs(graph, edge_attr, from = NULL, to = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
edge_attr	the name of the attribute for which to get values.
from	an optional vector of node IDs from which the edge is outgoing for filtering the list of edges.
to	an optional vector of node IDs from which the edge is incoming for filtering the list of edges.

Value

a named vector of edge attribute values for the attribute given by edge_attr by edge.

Examples

```
# Create a simple graph where
# edges have an edge attribute
# named `value`
graph <-
  create_graph() %>%
  add_n_nodes(n = 4) %>%
  {
    edges <-
      create_edge_df(
        from = c(1, 2, 1, 4),
          to = c(2, 3, 4, 3),
         rel = "rel")
    add_edge_df(
      graph = .,
      edge_df = edges)
  } %>%
  set_edge_attrs(
    edge_attr = value,
    values = 1.6,
    from = 1,
      to = 2) %>%
  set_edge_attrs(
    edge_attr = value,
    values = 4.3,
    from = 1,
      to = 4) %>%
  set_edge_attrs(
    edge_attr = value,
    values = 2.9,
    from = 2,
      to = 3) %>%
  set_edge_attrs(
    edge_attr = value,
    values = 8.4,
    from = 4,
      to = 3)
```

```
# Get the values for the
# `value` edge attribute
graph %>%
  get_edge_attrs(
    edge_attr = value)

# To only return edge attribute
# values for specified edges, use
# the `from` and `to` arguments
graph %>%
  get_edge_attrs(
    edge_attr = value,
    from = c(1, 2),
      to = c(2, 3))
```

`get_edge_attrs_ws` *Get edge attribute values*

Description

From a graph object of class `dgr_graph`, get edge attribute values for one or more edges.

Usage

```
get_edge_attrs_ws(graph, edge_attr)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`edge_attr`	the name of the attribute for which to get values.

Value

a named vector of edge attribute values for the attribute given by `edge_attr` by edge.

Examples

```
# Create a simple graph where
# edges have an edge attribute
# named `value`
graph <-
  create_graph() %>%
  add_n_nodes(n = 4) %>%
  {
    edges <-
      create_edge_df(
        from = c(1, 2, 1, 4),
          to = c(2, 3, 4, 3),
         rel = "rel")
```

```
    add_edge_df(
      graph = .,
      edge_df = edges)
  } %>%
  set_edge_attrs(
    edge_attr = value,
    values = 1.6,
    from = 1,
      to = 2) %>%
  set_edge_attrs(
    edge_attr = value,
    values = 4.3,
    from = 1,
      to = 4) %>%
  set_edge_attrs(
    edge_attr = value,
    values = 2.9,
    from = 2,
      to = 3) %>%
  set_edge_attrs(
    edge_attr = value,
    values = 8.4,
    from = 4,
      to = 3)

# Select the edges defined as
# `1`->`3` and `2`->`3`
graph <-
  graph %>%
  select_edges(
    from = c(1, 2),
    to = c(2, 3))

# Get the edge attribute values
# for the `value` attribute, limited
# to the current edge selection
graph %>%
  get_edge_attrs_ws(
    edge_attr = value)
```

`get_edge_count_w_multiedge` *Get count of edge definitions where multiple edges occur*

Description

Get a count of the number of edge definitions (e.g, '1' -> '2') where there are multiple edges (i.e., more than 1 edge of that definition, having distinct edge ID values). So, for example, if there are 2 edge definitions in the graph that involve 6 separate edge IDs (3 such edge IDs for each of the pairs of nodes), the count will be 2.

Usage

```
get_edge_count_w_multiedge(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a vector with a single, numerical value.

Examples

```
# Create a node data frame (ndf)
ndf <-
  create_node_df(
    n = 5,
    label = TRUE)

# Create an edge data frame (edf)
edf <-
  create_edge_df(
    from = c(1, 4, 4, 3, 5, 1, 3, 4),
      to = c(4, 1, 1, 2, 2, 2, 2, 1))

# Create a graph with the ndf and edf
graph <-
  create_graph(
    nodes_df = ndf,
    edges_df = edf)

# Get the total number of edge
# definitions (e.g., `4` -> `1`) where
# there are multiple edges (i.e.,
# distinct edges with separate edge
# ID values)
graph %>%
  get_edge_count_w_multiedge()
```

---

get_edge_df *Get an edge data frame from a graph*

Description

From a graph, obtain an edge data frame with all current edge attributes.

Usage

```
get_edge_df(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

an edge data frame.

Examples

```
# Create a graph
graph <-
  create_graph() %>%
  add_n_nodes(
    n = 1,
    type = "a") %>%
  select_last_nodes_created() %>%
  add_n_nodes_ws(
    n = 5,
    direction = "from",
    type = "b") %>%
  select_edges_by_node_id(
    nodes = 3:5) %>%
  set_edge_attrs_ws(
    edge_attr = color,
    value = "green") %>%
  set_edge_attrs_ws(
    edge_attr = rel,
    value = "a") %>%
  invert_selection %>%
  set_edge_attrs_ws(
    edge_attr = color,
    value = "blue") %>%
  set_edge_attrs_ws(
    edge_attr = rel,
    value = "b") %>%
  clear_selection()

# Get the graph's internal
# edge data frame (edf)
graph %>%
  get_edge_df()
```

---

get_edge_df_ws *Get the graph's edf filtered by a selection of edges*

Description

From a graph object of class dgr_graph, get the graph's internal edge data frame that is filtered by the edge ID values currently active as a selection.

Usage

```
get_edge_df_ws(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

an edge data frame.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 4,
    m = 4,
    set_seed = 23) %>%
  set_edge_attrs(
    edge_attr = value,
    values = c(2.5, 8.2, 4.2, 2.4))

# Select edges with ID values
# `1` and `3`
graph <-
  graph %>%
  select_edges_by_edge_id(
    edges = c(1, 3))

# Get the edge data frame that's
# limited to the rows that correspond
# to the edge selection
graph %>%
  get_edge_df_ws()
```

---

get_edge_ids *Get a vector of edge ID values*

Description

Obtain a vector of edge ID values from a graph object. An optional filter by edge attribute can limit the set of edge ID values returned.

Usage

```
get_edge_ids(graph, conditions = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
conditions	an option to use filtering conditions for the retrieval of edges.

Value

a vector of edge ID values.

Examples

```
# Create a node data frame (ndf)
ndf <-
  create_node_df(
    n = 4,
    type = "letter",
    color = c("red", "green", "grey", "blue"),
    value = c(3.5, 2.6, 9.4, 2.7))

# Create an edge data frame (edf)
edf <-
  create_edge_df(
    from = c(1, 2, 3),
    to = c(4, 3, 1),
    rel = "leading_to",
    color = c("pink", "blue", "blue"),
    value = c(3.9, 2.5, 7.3))

# Create a graph
graph <-
  create_graph(
    nodes_df = ndf,
    edges_df = edf)

# Get a vector of all edges in a graph
graph %>%
  get_edge_ids()

# Get a vector of edge ID values using a
# numeric comparison (i.e., all edges with
# `value` attribute greater than 3)
get_edge_ids(
  graph,
  conditions = value > 3)

# Get a vector of edge ID values using
# a match pattern (i.e., all edges with
# `color` attribute of `pink`)
get_edge_ids(
  graph,
  conditions = color == "pink")

# Use multiple conditions to return edges
```

```
# with the desired attribute values
get_edge_ids(
  graph,
  conditions =
    color == "blue" &
    value > 5)
```

`get_edge_info` *Get detailed information on edges*

Description

Obtain a data frame with detailed information on edges and their interrelationships within the graph.

Usage

```
get_edge_info(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a data frame containing information specific to each edge within the graph.

Examples

```
# Create a simple graph
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 5, m = 10,
    set_seed = 23)

# Get information on the
# graph's edges
graph %>%
  get_edge_info()
```

get_eigen_centrality *Get the eigen centrality for all nodes*

Description

Get the eigen centrality values for all nodes in the graph.

Usage

```
get_eigen_centrality(graph, weights_attr = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
weights_attr	an optional name of the edge attribute to use in the adjacency matrix. If NULL then, if it exists, the weight edge attribute of the graph will be used. If NA then no edge weights will be used.

Value

a data frame with eigen centrality scores for each of the nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 10, m = 15,
    set_seed = 23)

# Get the eigen centrality scores
# for nodes in the graph
graph %>%
  get_eigen_centrality()
```

get_girth *Get graph girth*

Description

Get the girth of a graph, which is the length of the shortest circle in the graph. Loop edges and multiple edges are not considered. If the graph contains no cycles then zero is returned.

Usage

```
get_girth(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a single numeric value representing the length of the shortest circle in the graph.

Examples

```
# Create a cycle graph
graph <-
  create_graph() %>%
  add_cycle(n = 5)

# Determine the graph's girth
graph %>%
  get_girth()

# Create a full graph and then
# get the girth for that
create_graph() %>%
  add_full_graph(n = 10) %>%
  get_girth()
```

---

get_global_graph_attr_info

Get global graph attributes

Description

Get the available global attributes for a graph object of class dgr_graph.

Usage

```
get_global_graph_attr_info(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a data frame containing global attributes for the graph.

Examples

```
# Create a new, empty graph
graph <- create_graph()

# View the graph's set of
# global attributes
graph %>%
  get_global_graph_attr_info()
```

---

get_graph_actions *Get information on any available graph actions*

Description

Get a tibble of the available graph actions, which contains information on function invocations to be called on the graph at every transformation step, or, when manually invoked with the `trigger_graph_actions()` function.

Usage

```
get_graph_actions(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a `df_tbl` object.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Add a graph action that sets a node
# attr column with a function; the
# main function `set_node_attr_w_fcn()`
# uses the `get_betweenness()` function
# to provide betweenness values in the
# `btwns` column
graph <-
```

```
  graph %>%
  add_graph_action(
    fcn = "set_node_attr_w_fcn",
    node_attr_fcn = "get_betweenness",
    column_name = "btwns",
    action_name = "get_btwns")

# To ensure that the action is
# available in the graph, use the
# `get_graph_actions()` function
graph %>%
  get_graph_actions()
```

`get_graph_from_graph_series` *Get a graph available in a series*

Description

Using a graph series object of type `dgr_graph_1D`, get a graph object.

Usage

```
get_graph_from_graph_series(graph_series, graph_no)
```

Arguments

`graph_series`	a graph series object of type `dgr_graph_1D`.
`graph_no`	the index of the graph in the graph series.

Examples

```
# Create three graphs
graph_1 <-
  create_graph() %>%
  add_path(n = 4)

graph_2 <-
  create_graph() %>%
  add_cycle(n = 5)

graph_3 <-
  create_graph() %>%
  add_star(n = 6)

# Create an empty graph series
# and add the graphs
series <-
  create_graph_series() %>%
```

```
  add_graph_to_graph_series(
    graph = graph_1) %>%
  add_graph_to_graph_series(
    graph = graph_2) %>%
  add_graph_to_graph_series(
    graph = graph_3)

# Get the second graph in the series
extracted_graph <-
  series %>%
  get_graph_from_graph_series(
    graph_no = 2)
```

`get_graph_info` *Get metrics for a graph*

Description

Get a data frame with metrics for a graph.

Usage

```
get_graph_info(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a data frame containing metrics pertaining to the graph

Examples

```
## Not run:
# Import a GML graph file available
# in the DiagrammeR package
karate_club <-
  system.file(
    "extdata", "karate.gml",
    package = "DiagrammeR") %>%
  import_graph() %>%
  set_graph_name("karate")

# Display a data frame with
# graph information
karate_club %>%
  get_graph_info()

## End(Not run)
```

get_graph_log *Get the graph log information*

Description

Get a tibble of the graph log, which contains information on the functions called on the graph that resulted in some transformation of the graph.

Usage

```
get_graph_log(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a df_tbl object.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function and
# delete 2 nodes from the graph
graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23) %>%
  delete_node(node = 5) %>%
  delete_node(node = 7)

# Get the graph log, which is a
# record of all graph transformations
graph %>%
  get_graph_log()
```

get_graph_name *Get graph name*

Description

Get the name of a graph object of class dgr_graph.

Usage

```
get_graph_name(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a single-length character vector with the assigned graph name. If a graph name has not been set, NA is returned.

Examples

```
# Create an empty graph
graph <- create_graph()

# Provide the new graph with a name
graph <-
  set_graph_name(
    graph,
    name = "the_name")

# Get the graph's name
graph %>%
  get_graph_name()
```

get_graph_series_info *Get information on a graph series*

Description

Obtain a data frame with information on the graphs within a graph series.

Usage

```
get_graph_series_info(graph_series)
```

Arguments

graph_series a graph series object of type dgr_graph_1D.

Value

a data frame containing information on the graphs within the supplied graph series.

Examples

```
# Create three graphs
graph_1 <-
  create_graph() %>%
  add_path(n = 4)

graph_2 <-
  create_graph() %>%
  add_cycle(n = 5)

graph_3 <-
  create_graph() %>%
  add_star(n = 6)

# Create an empty graph series
# and add the graphs
series <-
  create_graph_series() %>%
  add_graph_to_graph_series(
    graph = graph_1) %>%
  add_graph_to_graph_series(
    graph = graph_2) %>%
  add_graph_to_graph_series(
    graph = graph_3)

# Get information on the graphs in the series
series %>%
  get_graph_series_info()
```

---

get_graph_time *Get the graph date-time or timezone*

Description

Set the time and timezone for a graph object of class dgr_graph.

Usage

```
get_graph_time(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a single-length POSIXct vector with the assigned graph time.

Examples

```
# Create an empty graph and
# set the graph's time; if nothing
# is supplied for the `tz` argument,
# `GMT` is used as the time zone
graph <-
  create_graph() %>%
    set_graph_time(
      time = "2015-10-25 15:23:00")

# Get the graph's time as a POSIXct
# object using `get_graph_time()`
graph %>%
  get_graph_time()
```

---

get_jaccard_similarity

Get Jaccard similarity coefficient scores

Description

Get the Jaccard similarity coefficient scores for one or more nodes in a graph.

Usage

```
get_jaccard_similarity(graph, nodes = NULL, direction = "all",
  round_to = 3)
```

Arguments

graph	a graph object of class dgr_graph.
nodes	an optional vector of node IDs to consider for Jaccard similarity scores. If not supplied, then similarity scores will be provided for every pair of nodes in the graph.
direction	using all (the default), the function will ignore edge direction when determining scores for neighboring nodes. With out and in, edge direction for neighboring nodes will be considered.
round_to	the maximum number of decimal places to retain for the Jaccard similarity coefficient scores. The default value is 3.

Value

a matrix with Jaccard similarity values for each pair of nodes considered.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Get the Jaccard similarity
# values for nodes `5`, `6`,
# and `7`
graph %>%
  get_jaccard_similarity(
    nodes = 5:7)
```

---

get_last_edges_created

Get the last set of edges created in a graph

Description

Get the last edges that were created in a graph object of class `dgr_graph`. This function should ideally be used just after creating the edges.

Usage

```
get_last_edges_created(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a vector of edge ID values.

Examples

```
# Create a graph and add a cycle and then
# a tree in 2 separate function calls
graph <-
  create_graph() %>%
  add_cycle(
    n = 3,
    rel = "a") %>%
  add_balanced_tree(
    k = 2, h = 2,
    rel = "b")

# Get the last edges created (all edges
# from the tree)
graph %>%
  get_last_edges_created()
```

`get_last_nodes_created` *Get the last set of nodes created in a graph*

Description

Get the last nodes that were created in a graph object of class `dgr_graph`. Provides a vector of node ID values. This function should ideally be used just after creating the nodes.

Usage

```
get_last_nodes_created(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a vector of node ID values.

Examples

```
# Create a graph and add 4 nodes
# in 2 separate function calls
graph <-
  create_graph() %>%
  add_n_nodes(
    n = 2,
    type = "a",
    label = c("a_1", "a_2")) %>%
  add_n_nodes(
```

```
    n = 2,
    type = "b",
    label = c("b_1", "b_2"))

# Get the last nodes created (2 nodes
# from the last function call)
graph %>%
  get_last_nodes_created()
#> [1] 3 4
```

`get_leverage_centrality` *Get leverage centrality*

Description

Get the leverage centrality values for all nodes in the graph. Leverage centrality is a measure of the relationship between the degree of a given node and the degree of each of its neighbors, averaged over all neighbors. A node with negative leverage centrality is influenced by its neighbors, as the neighbors connect and interact with far more nodes. A node with positive leverage centrality influences its neighbors since the neighbors tend to have far fewer connections.

Usage

```
get_leverage_centrality(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a data frame with leverage centrality values for each of the nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Get leverage centrality values
# for all nodes in the graph
graph %>%
```

```
  get_leverage_centrality()

# Add the leverage centrality
# values to the graph as a
# node attribute
graph <-
  graph %>%
  join_node_attrs(
    df = get_leverage_centrality(.))

# Display the graph's node data frame
graph %>%
  get_node_df()
```

`get_max_eccentricity` *Get the maximum graph eccentricity*

Description

Get the diameter of a graph, which is the largest eccentricity in the graph. The graph eccentricity of a node is its shortest path from the farthest other node in the graph.

Usage

```
get_max_eccentricity(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a single numeric value representing the maximum eccentricity of the graph.

Examples

```
# Create a cycle graph
graph <-
  create_graph() %>%
  add_cycle(n = 5)

# Determine the graph's maximum
# eccentricity
graph %>%
  get_max_eccentricity()

# Create a full graph and then
# get the maximum eccentricity
# value for that
create_graph() %>%
```

```
  add_full_graph(n = 10) %>%
  get_max_eccentricity()
```

`get_mean_distance` *Get the mean distance*

Description

Get the mean distance of a graph, which is the average path length in the graph. This operates through calculation of the shortest paths between all pairs of nodes..

Usage

```
get_mean_distance(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a single numeric value representing the mean distance of the graph.

Examples

```
# Create a cycle graph
graph <-
  create_graph() %>%
  add_cycle(n = 5)

# Determine the mean distance
graph %>%
  get_mean_distance()

# Create a full graph and then
# get the mean distance value
create_graph() %>%
  add_full_graph(n = 10) %>%
  get_mean_distance()
```

`get_min_cut_between` *Get the minimum cut between source and sink nodes*

Description

Get the minimum cut between source and sink nodes. This is the minimum total capacity of edges needed for removal in order to eliminate all paths from the source and sink nodes.

Usage

```
get_min_cut_between(graph, from, to)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`from`	the node ID for the source node.
`to`	the node ID for the sink or target node.

Value

a single numeric value representing the minimum total edge capacity removed to disconnect the source and sink nodes.

Examples

```
# Set a seed
set.seed(23)

# Create a cycle graph
graph <-
  create_graph() %>%
  add_cycle(n = 5)

# Determine the minimum cut
# between nodes `1` and `4`
graph %>%
  get_min_cut_between(
    from = 1,
    to = 2)

# Create a cycle graph with
# randomized values given to all
# edges as the `capacity` attribute
graph_capacity <-
  create_graph() %>%
  add_cycle(n = 5) %>%
  select_edges() %>%
  set_edge_attrs_ws(
    edge_attr = capacity,
```

```
    value =
      rnorm(
        n = count_edges(.),
        mean = 5,
        sd = 1)) %>%
  clear_selection()

# Determine the minimum cut
# between nodes `1` and `4` for
# this graph, where `capacity`is
# set as an edge attribute
graph_capacity %>%
  get_min_cut_between(
    from = 1,
    to = 2)

# Create a full graph and then
# get the minimum cut requirement
# between nodes `2` and `8`
create_graph() %>%
  add_full_graph(n = 10) %>%
  get_min_cut_between(
    from = 2,
    to = 8)
```

`get_min_eccentricity` *Get the minimum graph eccentricity*

Description

Get the radius of a graph, which is the smallest eccentricity in the graph. The graph eccentricity of a node is its shortest path from the farthest other node in the graph.

Usage

```
get_min_eccentricity(graph, direction = "all")
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`direction`	using `all` (the default), the search will ignore edge direction while traversing through the graph. With `out`, measurements of paths will be from a node whereas with `in`, measurements of paths will be to a node.

Value

a single numeric value representing the minimum eccentricity of the graph.

Examples

```
# Create a cycle graph
graph <-
  create_graph() %>%
  add_cycle(n = 5)

# Determine the graph's minimum
# eccentricity
graph %>%
  get_min_eccentricity()

# Create a full graph and then
# get the minimum eccentricity
# value for that
create_graph() %>%
  add_full_graph(n = 10) %>%
  get_min_eccentricity()
```

---

get_multiedge_count *Get the count of multiple edges*

Description

Get a count of the number of multiple edges in the graph. Included in the count is the number of separate edges that share the same edge definition (i.e., same pair of nodes) across the entire graph. So, for example, if there are 2 edge definitions in the graph that involve 6 separate edge IDs, the count will be 4.

Usage

```
get_multiedge_count(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a vector with a single, numerical value.

Examples

```
# Create a node data frame (ndf)
ndf <-
  create_node_df(
    n = 5,
    label = TRUE)

# Create an edge data frame (edf)
```

```
edf <-
  create_edge_df(
    from = c(1, 4, 4, 3, 5, 1, 3, 4),
      to = c(4, 1, 1, 2, 2, 2, 2, 1))

# Create a graph with the ndf and edf
graph <-
  create_graph(
    nodes_df = ndf,
    edges_df = edf)

# Get the total number of multiple
# edges (those edges that share an
# edge definition) in the graph
graph %>%
  get_multiedge_count()
```

`get_nbrs` *Get all neighbors of one or more nodes*

Description

With one or more nodes, get the set of all neighboring nodes.

Usage

```
get_nbrs(graph, nodes)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`nodes`	a vector of node ID values.

Value

a vector of node ID values.

Examples

```
# Create a simple, directed graph with 5
# nodes and 4 edges
graph <-
  create_graph() %>%
  add_path(n = 5)

# Find all neighbor nodes for node `2`
graph %>%
  get_nbrs(nodes = 2)
```

```
# Find all neighbor nodes for nodes `1`
# and `5`
graph %>%
  get_nbrs(nodes = c(1, 5))

# Color node `3` with purple, get its
# neighbors and color those nodes green
graph <-
  graph %>%
  select_nodes_by_id(nodes = 3) %>%
  set_node_attrs_ws(
    node_attr = color,
    value = "purple") %>%
  clear_selection() %>%
  select_nodes_by_id(
    nodes = get_nbrs(
      graph = .,
      nodes = 3)) %>%
  set_node_attrs_ws(
    node_attr = color,
    value = "green")
```

`get_node_attrs` *Get node attribute values*

Description

From a graph object of class `dgr_graph`, get node attribute values for one or more nodes.

Usage

```
get_node_attrs(graph, node_attr, nodes = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`node_attr`	the name of the attribute for which to get values.
`nodes`	an optional vector of node IDs for filtering list of nodes present in the graph or node data frame.

Value

a named vector of node attribute values for the attribute given by `node_attr` by node ID.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 4,
    m = 4,
    set_seed = 23) %>%
  set_node_attrs(
    node_attr = value,
    values = c(2.5, 8.2, 4.2, 2.4))

# Get all of the values from
# the `value` node attribute
# as a named vector
graph %>%
  get_node_attrs(
    node_attr = value)

# To only return node attribute
# values for specified nodes,
# use the `nodes` argument
graph %>%
  get_node_attrs(
    node_attr = value,
    nodes = c(1, 3))
```

---

get_node_attrs_ws *Get node attribute values from a selection of nodes*

Description

From a graph object of class dgr_graph, get node attribute values from nodes currently active as a selection.

Usage

```
get_node_attrs_ws(graph, node_attr)
```

Arguments

graph	a graph object of class dgr_graph.
node_attr	the name of the attribute for which to get values.

Value

a named vector of node attribute values for the attribute given by node_attr by node ID.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 4,
    m = 4,
    set_seed = 23) %>%
  set_node_attrs(
    node_attr = value,
    values = c(2.5, 8.2, 4.2, 2.4))

# Select nodes with ID values
# `1` and `3`
graph <-
  graph %>%
  select_nodes_by_id(
    nodes = c(1, 3))

# Get the node attribute values
# for the `value` attribute, limited
# to the current node selection
graph %>%
  get_node_attrs_ws(
    node_attr = value)
```

---

get_node_df *Get a node data frame from a graph*

Description

From a graph, obtain a node data frame with all current node attributes.

Usage

```
get_node_df(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a node data frame.

Examples

```
# Create a graph
graph <-
  create_graph() %>%
  add_n_nodes(
    n = 1,
    type = "a") %>%
  select_last_nodes_created() %>%
  add_n_nodes_ws(
    n = 5,
    direction = "from",
    type = "b") %>%
  select_nodes_by_id(
    nodes = 1) %>%
  set_node_attrs_ws(
    node_attr = value,
    value = 25.3) %>%
  clear_selection() %>%
  select_nodes_by_id(
    nodes = 2:4) %>%
  set_node_attrs_ws(
    node_attr = color,
    value = "grey70") %>%
  invert_selection() %>%
  set_node_attrs_ws(
    node_attr = color,
    value = "grey80") %>%
  clear_selection()

# Get the graph's internal node
# data frame (ndf)
graph %>%
  get_node_df()
```

---

get_node_df_ws *Get the graph's ndf filtered by a selection of nodes*

Description

From a graph object of class dgr_graph, get the graph's internal node data frame that is filtered by the node ID values currently active as a selection.

Usage

```
get_node_df_ws(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a node data frame.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 4,
    m = 4,
    set_seed = 23) %>%
  set_node_attrs(
    node_attr = value,
    values = c(2.5, 8.2, 4.2, 2.4))

# Select nodes with ID values
# `1` and `3`
graph <-
  graph %>%
  select_nodes_by_id(
    nodes = c(1, 3))

# Get the node data frame that's
# limited to the rows that correspond
# to the node selection
graph %>%
  get_node_df_ws()
```

---

get_node_ids *Get a vector of node ID values*

Description

Obtain a vector of node ID values from a graph object. An optional filter by node attribute can limit the set of node ID values returned.

Usage

```
get_node_ids(graph, conditions = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
conditions	an option to use filtering conditions for the retrieval of nodes.

Value

a vector of node ID values.

Examples

```
# Create a node data
# frame (ndf)
ndf <-
  create_node_df(
    n = 4,
    type = "letter",
    color = c(
      "red", "green",
      "blue", "blue"),
    value = c(
      3.5, 2.6, 9.4, 2.7))

# Create a graph using
# the ndf
graph <-
  create_graph(
    nodes_df = ndf)

# Get a vector of all nodes in a graph
graph %>%
  get_node_ids()

# Get a vector of node ID values using a
# numeric comparison (i.e., all nodes with
# `value` attribute greater than 3)
graph %>%
  get_node_ids(
    conditions = value > 3)

# Get a vector of node ID values using
# a match pattern (i.e., all nodes with
# `color` attribute of `green`)
graph %>%
  get_node_ids(
    conditions = color == "green")

# Use multiple conditions to return nodes
# with the desired attribute values
graph %>%
  get_node_ids(
    conditions =
      color == "blue" &
      value > 5)
```

get_node_info *Get detailed information on nodes*

Description

Obtain a data frame with detailed information on nodes and their interrelationships within the graph.

Usage

```
get_node_info(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a data frame containing information specific to each node within the graph.

Examples

```
# Create a simple graph
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 5, m = 10,
    set_seed = 23)

# Get information on the
# graph's nodes
graph %>%
  get_node_info()
```

get_non_nbrs *Get non-neighbors of a node in a graph*

Description

Get the set of all nodes not neighboring a single graph node.

Usage

```
get_non_nbrs(graph, node)
```

Arguments

graph	a graph object of class dgr_graph.
node	a single-length vector containing a node ID value.

Value

a vector of node ID values.

Examples

```
# Create a simple, directed graph with 5
# nodes and 4 edges
graph <-
  create_graph() %>%
  add_path(n = 5)

# Find all non-neighbors of node `2`
graph %>%
  get_non_nbrs(node = 2)
```

---

get_pagerank *Get the PageRank values for all nodes*

Description

Get the PageRank values for all nodes in the graph.

Usage

```
get_pagerank(graph, directed = TRUE, damping = 0.85)
```

Arguments

graph	a graph object of class dgr_graph.
directed	if TRUE (the default) then directed paths will be considered for directed graphs. This is ignored for undirected graphs.
damping	the damping factor. The default value is set to 0.85.

Value

a data frame with PageRank values for each of the nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)
```

```
# Get the PageRank scores
# for all nodes in the graph
graph %>%
  get_pagerank()

# Colorize nodes according to their
# PageRank scores
graph <-
  graph %>%
  join_node_attrs(
    df = get_pagerank(graph = .)) %>%
  colorize_node_attrs(
    node_attr_from = pagerank,
    node_attr_to = fillcolor,
    palette = "RdYlGn")
```

`get_paths` *Get paths from a specified node in a directed graph*

Description

Obtain a list of all possible paths from a given node within a directed graph

Usage

```
get_paths(graph, from = NULL, to = NULL, shortest_path = FALSE,
  longest_path = FALSE, distance = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`from`	the node from which all paths will be determined.
`to`	the node to which all paths will be determined.
`shortest_path`	an option to return paths that are the shortest in the set of all determined paths.
`longest_path`	an option to return paths that are the longest in the set of all determined paths.
`distance`	a vector of integer values that specify which of the valid paths to return when filtering by distance.

Value

a list of paths, sorted by ascending traversal length, comprising vectors of node IDs in sequence of traversal through the graph.

Examples

```
# Create a simple graph
graph <-
  create_graph() %>%
  add_n_nodes(n = 8) %>%
  add_edge(from = 1, to = 2) %>%
  add_edge(from = 1, to = 3) %>%
  add_edge(from = 3, to = 4) %>%
  add_edge(from = 3, to = 5) %>%
  add_edge(from = 4, to = 6) %>%
  add_edge(from = 2, to = 7) %>%
  add_edge(from = 7, to = 5) %>%
  add_edge(from = 4, to = 8)

# Get a list of all paths outward from node `1`
graph %>%
  get_paths(from = 1)

# Get a list of all paths leading to node `6`
graph %>%
  get_paths(to = 6)

# Get a list of all paths from `1` to `5`
graph %>%
  get_paths(
   from = 1,
   to = 5)

# Get a list of all paths from `1` up to a distance
# of 2 node traversals
graph %>%
  get_paths(
    from = 1,
    distance = 2)

# Get a list of the shortest paths from `1` to `5`
get_paths(
  graph,
  from = 1,
  to = 5,
  shortest_path = TRUE)

# Get a list of the longest paths from `1` to `5`
get_paths(
  graph,
  from = 1,
  to = 5,
  longest_path = TRUE)
```

---

get_periphery *Get nodes that form the graph periphery*

Description

Get those nodes that are part of the graph periphery (i.e., have the maximum eccentricity in the graph).

Usage

```
get_periphery(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a vector of node IDs.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function and
# get the nodes in the graph periphery
create_graph() %>%
  add_gnm_graph(
    n = 28,
    m = 35,
    set_seed = 23) %>%
  get_periphery()
```

---

get_predecessors *Get node IDs for predecessor nodes to the specified node*

Description

Provides a vector of node IDs for all nodes that have a connection to the given node.

Usage

```
get_predecessors(graph, node)
```

Arguments

graph	a graph object of class dgr_graph.
node	a node ID for the selected node.

Value

a vector of node ID values.

Examples

```
# Set a seed
set.seed(23)

# Create a node data frame (ndf)
ndf <- create_node_df(n = 26)

# Create an edge data
# frame (edf)
edf <-
  create_edge_df(
    from = sample(
      1:26, replace = TRUE),
    to = sample(
      1:26, replace = TRUE))

# From the ndf and edf,
# create a graph object
graph <-
  create_graph(
    nodes_df = ndf,
    edges_df = edf)

# Get predecessors for node
# `23` in the graph
graph %>%
  get_predecessors(
    node = 23)

# If there are no predecessors,
# `NA` is returned
graph %>%
  get_predecessors(
    node = 26)
```

---

get_radiality *Get radiality centrality scores*

Description

Get the radiality centrality for all nodes in a graph. These scores describe the ease to which nodes can reach other nodes.

Usage

```
get_radiality(graph, direction = "all")
```

Arguments

graph	a graph object of class dgr_graph.
direction	using all (the default), the search will ignore edge direction while traversing through the graph. With out, measurements of paths will be from a node whereas with in, measurements of paths will be to a node.

Value

a data frame with radiality centrality scores for each of the nodes.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Get the radiality scores for nodes in the graph
graph %>%
  get_radiality()

# Add the radiality values
# to the graph as a node
# attribute
graph <-
  graph %>%
  join_node_attrs(
    df = get_radiality(.))

# Display the graph's node data frame
graph %>%
  get_node_df()
```

---

get_reciprocity *Get the graph reciprocity*

Description

Get the reciprocity of a directed graph. The reciprocity of a graph is the fraction of reciprocal edges (e.g., '1' -> '2' and '2' -> '1') over all edges available in the graph. Note that for an undirected graph, all edges are reciprocal. This function does not consider loop edges (e.g., '1' -> '1').

Usage

```
get_reciprocity(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a single, numerical value that is the ratio value of reciprocal edges over all graph edges.

Examples

```
# Define a graph where 2 edge definitions
# have pairs of reciprocal edges
graph <-
  create_graph() %>%
  add_cycle(n = 3) %>%
  add_node(
    from = 1,
      to = 1) %>%
  add_node(
    from = 1,
      to = 1)

# Get the graph reciprocity, which will
# be calculated as the ratio 4/7 (where
# 4 is the number reciprocating edges
# and 7 is the total number of edges
# in the graph)
graph %>%
  get_reciprocity()

# For an undirected graph, all edges
# are reciprocal, so the ratio will
# always be 1
graph %>%
  set_graph_undirected() %>%
  get_reciprocity()

# For a graph with no edges, the graph
# reciprocity cannot be determined (and
# the same NA result is obtained from an
# empty graph)
create_graph() %>%
  add_n_nodes(n = 5) %>%
  get_reciprocity()
```

---

get_selection *Get the current selection available in a graph object*

Description

Get the current selection of node IDs or edge IDs from a graph object of class dgr_graph.

Usage

```
get_selection(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a vector with the current selection of nodes or edges.

Examples

```
# Create a simple graph
graph <-
  create_graph() %>%
  add_path(n = 6)

# Select node `4`, then select
# all nodes a distance of 1 away
# from node `4`, and finally
# return the selection of nodes as
# a vector object
graph %>%
  select_nodes(nodes = 4) %>%
  select_nodes_in_neighborhood(
    node = 4,
    distance = 1) %>%
  get_selection()

# Select edges associated with
# node `4` and return the
# selection of edges
graph %>%
  select_edges_by_node_id(
    nodes = 4) %>%
  get_selection()
```

---

get_similar_nbrs *Get neighboring nodes based on node attribute similarity*

Description

With a graph a single node serving as the starting point, get those nodes in a potential neighborhood of nodes (adjacent to the starting node) that have a common or similar (within threshold values) node attribute to the starting node.

Usage

```
get_similar_nbrs(graph, node, node_attr, tol_abs = NULL, tol_pct = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
node	a single-length vector containing a node ID value.
node_attr	the name of the node attribute to use to compare with adjacent nodes.
tol_abs	if the values contained in the node attribute node_attr are numeric, one can optionally supply a numeric vector of length 2 that provides a lower and upper numeric bound as criteria for neighboring node similarity to the starting node.
tol_pct	if the values contained in the node attribute node_attr are numeric, one can optionally supply a numeric vector of length 2 that specifies lower and upper bounds as negative and positive percentage changes to the value of the starting node. These bounds serve as criteria for neighboring node similarity to the starting node.

Value

a vector of node ID values.

Examples

```
# Getting similar neighbors can
# be done through numerical comparisons;
# start by creating a random, directed
# graph with 18 nodes and 22 edges
# using the `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 18,
    m = 25,
    set_seed = 23) %>%
  set_node_attrs(
    node_attr = value,
    values = rnorm(
      n = count_nodes(.),
      mean = 5,
      sd = 1) %>% round(0))

# Starting with node `10`, we
# can test whether any nodes
# adjacent and beyond are
# numerically equivalent in `value`
graph %>%
  get_similar_nbrs(
    node = 10,
    node_attr = value)

# We can also set a tolerance
# for ascribing similarly by using
# either the `tol_abs` or `tol_pct`
# arguments (the first applies absolute
```

```
# lower and upper bounds from the
# value in the starting node and the
# latter uses a percentage difference
# to do the same); try setting `tol_abs`
# with a fairly large range to
# determine if several nodes can be
# selected
graph %>%
  get_similar_nbrs(
    node = 10,
    node_attr = value,
    tol_abs = c(1, 1))

# That resulted in a fairly large
# set of 4 neigboring nodes; for
# sake of example, setting the range
# to be very large will effectively
# return all nodes in the graph
# except for the starting node
graph %>%
  get_similar_nbrs(
    node = 10,
    node_attr = value,
    tol_abs = c(10, 10)) %>%
    length()
```

`get_successors` *Get node IDs for successor nodes to the specified node*

Description

Provides a vector of node IDs for all nodes that have a connection from the given node.

Usage

```
get_successors(graph, node)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`node`	a node ID for the selected node.

Value

a vector of node ID values.

Examples

```
# Set a seed
set.seed(23)

# Create a node data frame (ndf)
ndf <- create_node_df(n = 26)

# Create an edge data
# frame (edf)
edf <-
  create_edge_df(
    from = sample(
      1:26, replace = TRUE),
    to = sample(
      1:26, replace = TRUE))

# From the ndf and edf,
# create a graph object
graph <-
  create_graph(
    nodes_df = ndf,
    edges_df = edf)

# Get sucessors for node
# `4` in the graph
graph %>%
  get_successors(
    node = 4)

# If there are no successors,
# NA is returned
graph %>%
  get_successors(
    node = 1)
```

`get_s_connected_cmpts` *Get nodes within strongly connected components*

Description

Determine which nodes in a graph belong to different strongly connected components.

Usage

```
get_s_connected_cmpts(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a data frame with nodes and their membership in different strongly connected components.

Examples

```
set.seed(23)

# Create a graph with a random
# connection between 2 different
# node cycles
graph <-
  create_graph() %>%
  add_cycle(
    n = 3,
    type = "cycle_1") %>%
  add_cycle(
    n = 4,
    type = "cycle_2") %>%
  add_edge(
    from =
      get_node_ids(
        graph = .,
        conditions =
          type == "cycle_1") %>%
        sample(size = 1),
    to =
      get_node_ids(
        graph = .,
        conditions =
          type == "cycle_2") %>%
        sample(size = 1))

# Get the strongly connected
# components as a data frame of
# nodes and their groupings
graph %>%
  get_s_connected_cmpts()
```

`get_w_connected_cmpts` *Get all nodes associated with connected components*

Description

Determine which nodes in a graph belong to different weakly connected components (i.e., distinct sets of nodes with traversable paths to and from each node in the set).

Usage

```
get_w_connected_cmpts(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a data frame with nodes and their membership in different weakly connected components.

Examples

```
# Create a graph with 2 cycles
graph <-
  create_graph() %>%
  add_cycle(n = 4) %>%
  add_cycle(n = 3)

# Check if the graph is connected
graph %>%
  is_graph_connected()

# Get the graph's weakly-connected
# components
graph %>%
  get_w_connected_cmpts()
```

---

grViz *R + viz.js*

Description

Make diagrams in R using viz.js with infrastructure provided by htmlwidgets.

Usage

```
grViz(diagram = "", engine = "dot", allow_subst = TRUE, options = NULL,
  width = NULL, height = NULL)
```

Arguments

diagram	spec for a diagram as either text, filename string, or file connection.
engine	string for the Graphviz layout engine; can be dot (default), neato, circo, or twopi. For more information see https://github.com/mdaines/viz.js#usage.
allow_subst	a boolean that enables/disables substitution functionality.
options	parameters supplied to the htmlwidgets framework.
width	an optional parameter for specifying the width of the resulting graphic in pixels.
height	an optional parameter for specifying the height of the resulting graphic in pixels.

Value

An object of class `htmlwidget` that will intelligently print itself into HTML in a variety of contexts including the R console, within R Markdown documents, and within Shiny output bindings.

`grVizOutput` *Widget output function for use in Shiny*

Description

Widget output function for use in Shiny

Usage

```
grVizOutput(outputId, width = "100%", height = "400px")
```

Arguments

`outputId`	output variable to read from
`width`	a valid CSS unit for the width or a number, which will be coerced to a string and have `px` appended.
`height`	a valid CSS unit for the height or a number, which will be coerced to a string and have `px` appended.

Examples

```
## Not run:
library(shiny)
library(shinyAce)

ui = shinyUI(fluidPage(fluidRow(
  column(
    width=4
    , aceEditor("ace", selectionId = "selection", value="digraph {A;}")
  ),
  column(
    width = 6
    , grVizOutput('diagram' )
  )
)))

server = function(input, output) {
  output$diagram <- renderGrViz({
    grViz(
      input$ace
    )
  })

}
```

```
shinyApp(ui = ui, server = server)

## End(Not run)
```

`import_graph` *Import a graph from various graph formats*

Description

Import a variety of graphs from different graph formats and create a graph object.

Usage

```
import_graph(graph_file, file_type = NULL, edges_extra_attr_names = NULL,
  edges_extra_attr_coltypes = NULL)
```

Arguments

`graph_file`	a connection to a graph file. When provided as a path to a file, it will read the file from disk. Files starting with `http://`, `https://`, `ftp://`, or `ftps://` will be automatically downloaded.
`file_type`	the type of file to be imported. Options are: `gml` (GML), `sif` (SIF), `edges` (a .edges file), and `mtx` (MatrixMarket format). If not supplied, the type of graph file will be inferred by its file extension.
`edges_extra_attr_names`	for `edges` files, a vector of attribute names beyond the `from` and `to` data columns can be provided in the order they appear in the input data file.
`edges_extra_attr_coltypes`	for `edges` files, this is a string of column types for any attribute columns provided for `edges_extra_attr_names`. This string representation is where each character represents each of the extra columns of data and the mappings are: `c` -> character, `i` -> integer, `n` -> number, `d` -> double, `l` -> logical, `D` -> date, `T` -> date time, `t` -> time, `?` -> guess, or `_/-`, which skips the column.

Value

a graph object of class `dgr_graph`.

Examples

```
## Not run:
# Import a GML graph file
gml_graph <-
  import_graph(
    system.file(
      "extdata/karate.gml",
      package = "DiagrammeR"))
```

```
# Get a count of the graph's nodes
gml_graph %>%
  count_nodes()

# Get a count of the graph's edges
gml_graph %>%
  count_edges()

## End(Not run)
```

`invert_selection` *Invert selection of nodes or edges in a graph*

Description

Modify the selection of nodes or edges within a graph object such that all nodes or edges previously not selected will now be selected and vice versa.

Usage

```
invert_selection(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a node data frame (ndf)
ndf <-
  create_node_df(
    n = 4,
    type = "standard")

# Create an edge data frame (edf)
edf <-
  create_edge_df(
    from = c(1, 2, 3),
    to = c(4, 3, 1),
    rel = "leading_to")

# Create a graph
graph <-
  create_graph(
    nodes_df = ndf,
```

```
    edges_df = edf)

# Select nodes with ID
# values `1` and `3`
graph <-
  graph %>%
  select_nodes(
    nodes = c(1, 3))

# Verify that a node
# selection has been made
graph %>%
  get_selection()

# Invert the selection
graph <-
  graph %>%
  invert_selection()

# Verify that the node
# selection has been changed
graph %>%
  get_selection()
```

`is_edge_loop` *Is the edge a loop edge?*

Description

Determines whether an edge definition is a loop edge.

Usage

```
is_edge_loop(graph, edge)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`edge`	a numeric edge ID value.

Value

a logical value.

Examples

```
# Create a graph that has multiple
# loop edges
graph <-
  create_graph() %>%
  add_path(n = 4) %>%
  add_edge(
    from = 1,
    to = 1) %>%
  add_edge(
    from = 3,
    to = 3)

# Get the graph's internal
# edge data frame
graph %>%
  get_edge_df()

# Determine if edge `4` is
# a loop edge
graph %>%
  is_edge_loop(edge = 4)

# Determine if edge `2` is
# a loop edge
graph %>%
  is_edge_loop(edge = 2)
```

---

is_edge_multiple *Is the edge a multiple edge?*

Description

Determines whether an edge definition has multiple edge IDs associated with the same node pair.

Usage

```
is_edge_multiple(graph, edge)
```

Arguments

graph	a graph object of class dgr_graph.
edge	a numeric edge ID value.

Value

a logical value.

Examples

```
# Create a graph that has multiple
# edges across some node pairs
graph <-
  create_graph() %>%
  add_path(n = 4) %>%
  add_edge(
    from = 1,
    to = 2) %>%
  add_edge(
    from = 3,
    to = 4)

# Get the graph's internal
# edge data frame
graph %>%
  get_edge_df()

# Determine if edge `1` is
# a multiple edge
graph %>%
  is_edge_multiple(edge = 1)

# Determine if edge `2` is
# a multiple edge
graph %>%
  is_edge_multiple(edge = 2)
```

---

is_edge_mutual *Is the edge mutual with another edge?*

Description

Determines whether an edge definition has a mutual analogue with the same node pair.

Usage

```
is_edge_mutual(graph, edge)
```

Arguments

graph	a graph object of class dgr_graph.
edge	a numeric edge ID value.

Value

a logical value.

Examples

```
# Create a graph that has mutual
# edges across some node pairs
graph <-
  create_graph() %>%
  add_path(n = 4) %>%
  add_edge(
    from = 4,
    to = 3) %>%
  add_edge(
    from = 2,
    to = 1)

# Get the graph's internal
# edge data frame
graph %>%
  get_edge_df()

# Determine if edge `1` has
# a mutual edge
graph %>%
  is_edge_mutual(edge = 1)

# Determine if edge `2` has
# a mutual edge
graph %>%
  is_edge_mutual(edge = 2)
```

---

is_edge_present *Determine whether a specified edge is present*

Description

From a graph object of class dgr_graph, determine whether an edge (defined by a pair of node IDs or node label values) is present.

Usage

```
is_edge_present(graph, edge = NULL, from = NULL, to = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
edge	an edge ID value to test for presence in the graph. If a single, numeric value is provided then values for from or to needn't be supplied.
from	a node ID from which the edge is outgoing, or, the label associated with the node. For an undirected graph, the value in from can be interchangeable with that in to.

`to` a node ID to which the edge is incoming, or, the label associated with the node. For an undirected graph, the value in `to` can be interchangeable with that in `from`.

Value

a logical value.

Examples

```
# Create a simple graph with
# a path of four nodes
graph <-
  create_graph() %>%
  add_path(
    n = 4,
    type = "path",
    label = c("one", "two",
              "three", "four"))

# Find out if edge ID `3`
# is present in the graph
graph %>%
  is_edge_present(edge = 3)

# Determine if there are any edges
# with the definition `1` -> `2`
graph %>%
  is_edge_present(
    from = 1,
    to = 2)

# Determine if there are any edges
# with the definition `4` -> `5`
graph %>%
  is_edge_present(
    from = 4,
    to = 5)

# Determine whether an edge,
# defined by its labels as
# `two` -> `three`, exists in
# the graph
graph %>%
  is_edge_present(
    from = "two",
    to = "three")

# Set the graph as undirected
# and determine whether an
# edge between nodes with labels
# `three` and `two` exists
```

```
graph %>%
  set_graph_undirected() %>%
  is_edge_present(
    from = "three",
    to = "two")
```

`is_graph_connected` *Is the graph a connected graph?*

Description

Determines whether a graph is a connected graph.

Usage

```
is_graph_connected(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a logical value.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function; this
# graph is not connected
create_graph() %>%
  add_gnm_graph(
    n = 15,
    m = 10,
    set_seed = 23) %>%
  is_graph_connected()

# Create another random graph;
# this graph is connected
create_graph() %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23) %>%
  is_graph_connected()
```

`is_graph_dag` *Is the graph a directed acyclic graph?*

Description

Provides a logical value on whether the graph is a directed acyclic graph (DAG). The conditions for a graph that is a DAG are that it should be a directed graph and it should not contain any cycles.

Usage

```
is_graph_dag(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a logical value.

Examples

```
# Create a directed graph containing
# only a balanced tree
graph_tree <-
  create_graph() %>%
  add_balanced_tree(
    k = 2, h = 3)

# Determine whether this graph
# is a DAG
graph_tree %>%
  is_graph_dag()

# Create a directed graph containing
# a single cycle
graph_cycle <-
  create_graph() %>%
  add_cycle(n = 5)

# Determine whether this graph
# is a DAG
graph_cycle %>%
  is_graph_dag()

# Create an undirected graph
# containing a balanced tree
graph_tree_undirected <-
  create_graph(
    directed = FALSE) %>%
```

```
  add_balanced_tree(
    k = 2, h = 2)

# Determine whether this graph
# is a DAG
graph_tree_undirected %>%
  is_graph_dag()
```

`is_graph_directed` *Is the graph a directed graph?*

Description

Determines whether a graph is set to be directed or not and returns a logical value to that effect.

Usage

```
is_graph_directed(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a logical value.

Examples

```
# Create an empty graph; by default,
# new graphs made by `create_graph()`
# are directed
graph <-
  create_graph()

# Determine whether the graph
# is directed
graph %>%
  is_graph_directed()

# Use the `set_graph_undirected()`
# function and check again whether
# the graph is directed
graph %>%
  set_graph_undirected() %>%
  is_graph_directed()
```

is_graph_empty *Is the graph empty?*

Description

Provides a logical value on whether the graph is empty (i.e., contains no nodes).

Usage

```
is_graph_empty(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a logical value.

Examples

```
# Create an empty graph
graph <- create_graph()

# Determine whether the graph is empty
graph %>%
  is_graph_empty()

# Create a non-empty graph
graph <-
  create_graph() %>%
  add_n_nodes(n = 3)

# Determine whether this graph is empty
graph %>%
  is_graph_empty()
```

is_graph_simple *Is the graph a simple graph?*

Description

Determine whether the graph is a simple graph. A simple graph is one that does not contain any loops nor any multiple edges.

Usage

```
is_graph_simple(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a logical value.

Examples

```
# Create a graph with 2 cycles
graph <-
  create_graph() %>%
  add_cycle(n = 4) %>%
  add_cycle(n = 3)

# Check if the graph is simple
graph %>%
  is_graph_simple()
```

---

is_graph_undirected *Is the graph an undirected graph?*

Description

Determines whether a graph is set as undirected or not and returns a logical value to that effect.

Usage

```
is_graph_undirected(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a logical value.

Examples

```
# Create an empty graph; by
# default, new graphs made
# by `create_graph()` are
# directed graph, so, use
# `directed = FALSE` to create
# an undirected graph
graph <-
  create_graph(
    directed = FALSE)
```

```
# Determine whether the
# graph is undirected
graph %>%
  is_graph_undirected()

# Use the `set_graph_directed()`
# function and check again
# as to whether the graph is
# undirected
graph %>%
  set_graph_directed() %>%
  is_graph_undirected()
```

`is_graph_weighted` *Is the graph a weighted graph?*

Description

Provides a logical value on whether the graph is weighted. A graph is considered to be weighted when it contains edges that all have a edge `weight` attribute with numerical values assigned for all edges.

Usage

```
is_graph_weighted(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a logical value.

Examples

```
# Create a graph where the edges have
# a `weight` attribute
graph <-
  create_graph() %>%
  add_cycle(n = 5) %>%
  select_edges() %>%
  set_edge_attrs_ws(
    edge_attr = weight,
    value = c(3, 5, 2, 9, 6)) %>%
  clear_selection()

# Determine whether the graph
# is a weighted graph
```

```
graph %>%
  is_graph_weighted()

# Create graph where the edges do
# not have a `weight` attribute
graph <-
  create_graph() %>%
  add_cycle(n = 5)

# Determine whether this graph
# is weighted
graph %>%
  is_graph_weighted()
```

`is_node_present` *Determine whether a specified node is present*

Description

From a graph object of class `dgr_graph`, determine whether a specified node is present.

Usage

```
is_node_present(graph, node)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`node`	either a node ID value or a node label to test for presence in the graph.

Value

a logical value.

Examples

```
# Create a simple graph with
# a path of four nodes
graph <-
  create_graph() %>%
  add_path(
    n = 4,
    type = "path",
    label = c(
      "one", "two",
      "three", "four"))

# Determine if there is a node
# with ID `1` in the graph
```

```
graph %>%
  is_node_present(node = 1)

# Determine if there is a node
# with ID `5` in the graph
graph %>%
  is_node_present(node = 5)

# Determine if there is a node
# with label `two` in the graph
graph %>%
  is_node_present(node = "two")
```

`is_property_graph` *Is the graph a property graph?*

Description

Provides a logical value on whether the graph is property graph (i.e., all nodes have an assigned `type` value and all edges have an assigned `rel` value).

Usage

```
is_property_graph(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a logical value. # Create a graph with 2 nodes # (with ‘type‘ values) and a # single edge (with a ‘rel‘) simple_property_graph <- create_graph() add_node(type = "a", label = "first") add_node(type = "b", label = "second") add_edge(from = "first", to = "second", rel = "rel_1")

This is indeed a property graph # but to confirm this, use the # ‘is_property_graph()‘ function is_property_graph(simple_property_graph)

If a ‘type‘ attribute is # removed, then this graph will # no longer be a property graph simple_property_graph set_node_attrs(node_attr = type, values = NA, nodes = 1) is_property_graph()

An empty graph will return FALSE create_graph() is_property_graph()

`join_edge_attrs` *Join new edge attribute values using a data frame*

Description

Join new edge attribute values in a left join using a data frame. The use of a left join in this function allows for no possibility that edges in the graph might be removed after the join.

Usage

```
join_edge_attrs(graph, df, by_graph = NULL, by_df = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`df`	the data frame to use for joining.
`by_graph`	optional specification of the column in the graph's internal edge data frame for the left join. If both `by_graph` and `by_df` are not provided, then a natural join will occur if there are columns in the graph's edf and in `df` with identical names.
`by_df`	optional specification of the column in `df` for the left join. If both `by_graph` and `by_df` are not provided, then a natural join will occur if there are columns in the graph's edf and in `df` with identical names.

Value

a graph object of class `dgr_graph`.

Examples

```
# Set a seed
set.seed(23)

# Create a simple graph
graph <-
  create_graph() %>%
  add_n_nodes(n = 5) %>%
  add_edges_w_string(
    edges = "1->2 1->3 2->4 2->5 3->5")

# Create a data frame with node ID values
# representing the graph edges (with `from` and `to`
# columns), and, a set of numeric values
df <-
  data.frame(
    from = c(1, 1, 2, 2, 3),
    to = c(2, 3, 4, 5, 5),
    values = rnorm(5, 5))
```

```
# Join the values in the data frame to the
# graph's edges; this works as a left join using
# identically-named columns in the graph and the df
# (in this case `from` and `to` are common to both)
graph <-
  graph %>%
  join_edge_attrs(
    df = df)

# Get the graph's internal edf to show that the
# join has been made
graph %>%
  get_edge_df()
```

`join_node_attrs` *Join new node attribute values using a data frame*

Description

Join new node attribute values in a left join using a data frame. The use of a left join in this function allows for no possibility that nodes in the graph might be removed after the join.

Usage

```
join_node_attrs(graph, df, by_graph = NULL, by_df = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`df`	the data frame to use for joining.
`by_graph`	optional specification of the column in the graph's internal node data frame for the left join. If both `by_graph` and `by_df` are not provided, then a natural join will occur if there are columns in the graph's ndf and in `df` with identical names.
`by_df`	optional specification of the column in `df` for the left join. If both `by_graph` and `by_df` are not provided, then a natural join will occur if there are columns in the graph's ndf and in `df` with identical names. `dgr_graph` that is created using `create_graph`.

Value

a graph object of class `dgr_graph`.

Examples

```
# Set a seed
set.seed(23)

# Create a simple graph
graph <-
  create_graph() %>%
  add_n_nodes(n = 5) %>%
  add_edges_w_string(
    edges = "1->2 1->3 2->4 2->5 3->5")

# Create a data frame with node ID values and a
# set of numeric values
df <-
  data.frame(
    values = round(rnorm(6, 5), 2),
    id = 1:6)

# Join the values in the data frame to the
# graph's nodes; this works as a left join using
# identically-named columns in the graph and the df
# (in this case the `id` column is common to both)
graph <-
  graph %>%
  join_node_attrs(
    df = df)

# Get the graph's internal ndf to show that the
# join has been made
graph %>%
  get_node_df()

# Get betweenness values for each node and
# add them as a node attribute (Note the
# common column name `id` in the different
# tables results in a natural join)
graph <-
  graph %>%
  join_node_attrs(
    df = get_betweenness(.))

# Get the graph's internal ndf to show that
# this join has been made
graph %>%
  get_node_df()
```

`layout_nodes_w_string` *Layout nodes using a text-based schematic*

Description

Layout one or several groups of nodes using a text-based schematic. The option is available to apply sorting to each of the groups.

Usage

```
layout_nodes_w_string(graph, layout, nodes, sort = NULL, width = 8,
  height = 8, ll = c(0, 0))
```

Arguments

graph	a graph object of class dgr_graph.
layout	a layout character string that provides a schematic for the layout. This consists of a rectangular collection of - characters (for no node placement), and numbers from 1 to 9 (representing different groupings of nodes, further described in the nodes argument).
nodes	a named vector of the form: c("1" = "[node_attr]:[value]", ...). The LHS corresponds to the numbers used in the layout schematic. The RHS provides a shorthand for the node attribute and a value for grouping together nodes (separated by a colon). For instance, with "type:a" in the RHS (and "1" in the LHS) we would target all nodes with a type attribute equal to a for positioning in the graph as described by the 1s in the layout.
sort	an optional sorting method to apply to the collection of nodes before assigning positional information. Like nodes, this is a named vector of the form: c("1" = "[node_attr]:asc|desc", ...). The node_attr in this case should be different than that used in nodes. Ideally, this node attribute should have unique values. Choose either asc or desc right of the colon for ascending or descending sorts.
width	the width of the layout diagram.
height	the height of the layout diagram.
ll	a vector describing the the lower-left coordinates of the layout

Value

a graph object of class dgr_graph.

Examples

```
# Create a graph with unique labels and
# several node `type` groupings
graph <-
  create_graph() %>%
  add_node(type = "a", label = "a") %>%
  add_node(type = "a", label = "b") %>%
  add_node(type = "b", label = "c") %>%
  add_node(type = "b", label = "d") %>%
  add_node(type = "b", label = "e") %>%
  add_node(type = "c", label = "f") %>%
```

```
  add_node(type = "c", label = "g")

# Define a 'layout' for groups of nodes
# using a text string (dashes are empty
# grid cells, numbers--representing
# ad-hoc groupings--correspond to
# individual nodes); here, define a layout
# with 3 groups of nodes
layout <-
"
1--------
1--------
---222---
--------3
--------3
"

# Use the `layout` along with what nodes
# the numbers correspond to in the graph
# with the `nodes` named vectors; the
# optional `sort` vector describes how
# we should sort the collection of node
# before adding position information
graph <-
  graph %>%
  layout_nodes_w_string(
    layout = layout,
    nodes = c("1" = "type:a",
              "2" = "type:b",
              "3" = "type:c"),
    sort = c("1" = "label:asc",
             "2" = "label:desc",
             "3" = "label:desc"))

# Show the graph's node data frame
# to confirm that `x` and `y` values
# were added to each of the nodes
graph %>%
  get_node_df()
```

`mermaid` *R + mermaid.js*

Description

Make diagrams in R using mermaid.js with infrastructure provided by htmlwidgets.

Usage

```
mermaid(diagram = "", ..., width = NULL, height = NULL)
```

Arguments

diagram	diagram in mermaid markdown-like language or file (as a connection or file name) containing a diagram specification. If no diagram is provided `diagram = ""` then the function will assume that a diagram will be provided by `tags` and `DiagrammeR` is just being used for dependency injection.
...	other arguments and parameters you would like to send to Javascript.
width	the width of the resulting graphic in pixels.
height	the height of the resulting graphic in pixels.

Value

An object of class `htmlwidget` that will intelligently print itself into HTML in a variety of contexts including the R console, within R Markdown documents, and within Shiny output bindings.

Examples

```
## Not run:
# Create a simple graph running left to right (note
# that the whitespace is not important)
DiagrammeR("
  graph LR
    A-->B
    A-->C
    C-->E
    B-->D
    C-->D
    D-->F
    E-->F
")
# Create the equivalent graph but have it running
# from top to bottom
DiagrammeR("
   graph TB
   A-->B
   A-->C
   C-->E
   B-->D
   C-->D
   D-->F
   E-->F
")

# Create a graph with different node shapes and
# provide fill styles for each node
DiagrammeR("graph LR;A(Rounded)-->B[Squared];B-->C{A Decision};
 C-->D[Square One];C-->E[Square Two];
 style A fill:#E5E25F;  style B fill:#87AB51; style C fill:#3C8937;
 style D fill:#23772C;  style E fill:#B6E6E6;"
)
```

```
# Load in the 'mtcars' dataset
data(mtcars)
connections <- sapply(
 1:ncol(mtcars)
  ,function(i) {
     paste0(
        i
      ,"(",colnames(mtcars)[i],")---"
      ,i,"-stats("
      ,paste0(
        names(summary(mtcars[,i]))
        ,": "
        ,unname(summary(mtcars[,i]))
        ,collapse="<br/>"
      )
      ,")"
   )
 }
)

# Create a diagram using the 'connections' object
DiagrammeR(
   paste0(
     "graph TD;", "\n",
     paste(connections, collapse = "\n"),"\n",
     "classDef column fill:#0001CC, stroke:#0D3FF3, stroke-width:1px;" ,"\n",
     "class ", paste0(1:length(connections), collapse = ","), " column;"
   )
 )

# Also with \code{DiagrammeR()}, you can use tags
# from \code{htmltools} (just make sure to use
# \code{class = "mermaid"})
library(htmltools)
diagramSpec = "
graph LR;
  id1(Start)-->id2(Stop);
  style id1 fill:#f9f,stroke:#333,stroke-width:4px;
  style id2 fill:#ccf,stroke:#f66,stroke-width:2px,stroke-dasharray: 5, 5;
"
html_print(tagList(
  tags$h1("R + mermaid.js = Something Special")
  ,tags$pre(diagramSpec)
  ,tags$div(class="mermaid",diagramSpec)
  ,DiagrammeR()
))

# Create a sequence diagram
DiagrammeR("
sequenceDiagram;
   customer->>ticket seller: ask for a ticket;
   ticket seller->>database: seats;
   alt tickets available
```

```
    database->>ticket seller: ok;
    ticket seller->>customer: confirm;
    customer->>ticket seller: ok;
    ticket seller->>database: book a seat;
    ticket seller->>printer: print a ticket;
  else sold out
    database->>ticket seller: none left;
    ticket seller->>customer: sorry;
  end
")

## End(Not run)
```

`mutate_edge_attrs` *Mutate a set of edge attribute values*

Description

Within a graph's internal edge data frame (edf), mutate numeric edge attribute values using one or more expressions.

Usage

```
mutate_edge_attrs(graph, ...)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`...`	expressions used for the mutation of edge attributes. LHS of each expression is either an existing or new edge attribute name. The RHS can consist of any valid R code that uses edge attributes as variables. Expressions are evaluated in the order provided, so, edge attributes created or modified are ready to use in subsequent expressions.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a graph with 3 edges
graph <-
  create_graph() %>%
  add_path(n = 4) %>%
  set_edge_attrs(
    edge_attr = width,
    values = c(3.4, 2.3, 7.2))

# Get the graph's internal edf
```

```
# to show which edge attributes
# are available
graph %>%
  get_edge_df()

# Mutate the `width` edge
# attribute, dividing each
# value by 2
graph <-
  graph %>%
  mutate_edge_attrs(
    width = width / 2)

# Get the graph's internal
# edf to show that the edge
# attribute `width` had its
# values changed
graph %>%
  get_edge_df()

# Create a new edge attribute,
# called `length`, that is the
# log of values in `width` plus
# 2 (and, also, round all values
# to 2 decimal places)
graph <-
  graph %>%
  mutate_edge_attrs(
    length = (log(width) + 2) %>%
                round(2))

# Get the graph's internal edf
# to show that the edge attribute
# values had been mutated
graph %>%
  get_edge_df()

# Create a new edge attribute
# called `area`, which is the
# product of the `width` and
# `length` attributes
graph <-
  graph %>%
  mutate_edge_attrs(
    area = width * length)

# Get the graph's internal edf
# to show that the edge attribute
# values had been multiplied
# together (with new attr `area`)
graph %>%
  get_edge_df()
```

`mutate_edge_attrs_ws` *Mutate edge attribute values for a selection of edges*

Description

Within a graph's internal edge data frame (edf), mutate edge attribute values only for edges in a selection by using one or more expressions.

Usage

```
mutate_edge_attrs_ws(graph, ...)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`...`	expressions used for the mutation of edge attributes. LHS of each expression is either an existing or new edge attribute name. The RHS can consist of any valid R code that uses edge attributes as variables. Expressions are evaluated in the order provided, so, edge attributes created or modified are ready to use in subsequent expressions.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a graph with 3 edges
# and then select edge `1`
graph <-
  create_graph() %>%
  add_path(n = 4) %>%
  set_edge_attrs(
    edge_attr = width,
    values = c(3.4, 2.3, 7.2)) %>%
  select_edges(edges = 1)

# Get the graph's internal edf
# to show which edge attributes
# are available
graph %>%
  get_edge_df()

# Mutate the `width` edge
# attribute for the edges
# only in the active selection
# of edges (edge `1`); here,
# we divide each value in the
# selection by 2
```

```
graph <-
  graph %>%
  mutate_edge_attrs_ws(
    width = width / 2)

# Get the graph's internal
# edf to show that the edge
# attribute `width` had its
# values changed
graph %>%
  get_edge_df()

# Create a new edge attribute,
# called `length`, that is the
# log of values in `width` plus
# 2 (and, also, round all values
# to 2 decimal places)
graph <-
  graph %>%
  clear_selection() %>%
  select_edges(edges = 2:3) %>%
  mutate_edge_attrs_ws(
    length = (log(width) + 2) %>%
               round(2))

# Get the graph's internal edf
# to show that the edge attribute
# values had been mutated only
# for edges `2` and `3` (since
# edge `1` is excluded, an NA
# value is applied)
graph %>%
  get_edge_df()

# Create a new edge attribute
# called `area`, which is the
# product of the `width` and
# `length` attributes
graph <-
  graph %>%
  mutate_edge_attrs_ws(
    area = width * length)

# Get the graph's internal edf
# to show that the edge attribute
# values had been multiplied
# together (with new attr `area`)
# for nodes `2` and `3`
graph %>%
  get_edge_df()

# We can invert the selection
# and mutate edge `1` several
```

```
# times to get an `area` value
# for that edge
graph <-
  graph %>%
  invert_selection() %>%
  mutate_edge_attrs_ws(
    length = (log(width) + 5) %>%
               round(2),
    area = width * length)

# Get the graph's internal edf
# to show that the 2 mutations
# occurred for edge `1`, yielding
# non-NA values for its edge
# attributes without changing
# those of the other edges
graph %>%
  get_edge_df()
```

`mutate_node_attrs` *Mutate a set of node attribute values*

Description

Within a graph's internal node data frame (ndf), mutate numeric node attribute values using one or more expressions.

Usage

```
mutate_node_attrs(graph, ...)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`...`	expressions used for the mutation of node attributes. LHS of each expression is either an existing or new node attribute name. The RHS can consist of any valid R code that uses node attributes as variables. Expressions are evaluated in the order provided, so, node attributes created or modified are ready to use in subsequent expressions.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a graph with 3 nodes
graph <-
  create_graph() %>%
  add_path(n = 3) %>%
  set_node_attrs(
    node_attr = width,
    values = c(1.4, 0.3, 1.1))

# Get the graph's internal ndf
# to show which node attributes
# are available
graph %>%
  get_node_df()

# Mutate the `width` node
# attribute, dividing each
# value by 2
graph <-
  graph %>%
  mutate_node_attrs(
    width = width / 2)

# Get the graph's internal
# ndf to show that the node
# attribute `width` had its
# values changed
graph %>%
  get_node_df()

# Create a new node attribute,
# called `length`, that is the
# log of values in `width` plus
# 2 (and, also, round all values
# to 2 decimal places)
graph <-
  graph %>%
  mutate_node_attrs(
    length = (log(width) + 2) %>%
               round(2))

# Get the graph's internal ndf
# to show that the node attribute
# values had been mutated
graph %>%
  get_node_df()

# Create a new node attribute
# called `area`, which is the
# product of the `width` and
# `length` attributes
graph <-
```

```
  graph %>%
  mutate_node_attrs(
    area = width * length)

# Get the graph's internal ndf
# to show that the node attribute
# values had been multiplied
# together (with new attr `area`)
graph %>%
  get_node_df()
```

`mutate_node_attrs_ws` *Mutate node attribute values for a selection of nodes*

Description

Within a graph's internal node data frame (ndf), mutate node attribute values only for nodes in a selection by using one or more expressions.

Usage

```
mutate_node_attrs_ws(graph, ...)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`...`	expressions used for the mutation of node attributes. LHS of each expression is either an existing or new node attribute name. The RHS can consist of any valid R code that uses node attributes as variables. Expressions are evaluated in the order provided, so, node attributes created or modified are ready to use in subsequent expressions.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a graph with 3 nodes
# and then select node `1`
graph <-
  create_graph() %>%
  add_path(n = 3) %>%
  set_node_attrs(
    node_attr = width,
    values = c(1.4, 0.3, 1.1)) %>%
  select_nodes(nodes = 1)

# Get the graph's internal ndf
```

```
# to show which node attributes
# are available
graph %>%
  get_node_df()

# Mutate the `width` node
# attribute for the nodes
# only in the active selection
# of nodes (node `1`); here,
# we divide each value in the
# selection by 2
graph <-
  graph %>%
  mutate_node_attrs_ws(
    width = width / 2)

# Get the graph's internal
# ndf to show that the node
# attribute `width` was
# mutated only for node `1`
graph %>%
  get_node_df()

# Create a new node attribute,
# called `length`, that is the
# log of values in `width` plus
# 2 (and, also, round all values
# to 2 decimal places)
graph <-
  graph %>%
  clear_selection() %>%
  select_nodes(nodes = 2:3) %>%
  mutate_node_attrs_ws(
    length = (log(width) + 2) %>%
               round(2))

# Get the graph's internal ndf
# to show that the node attribute
# values had been mutated only
# for nodes `2` and `3` (since
# node `1` is excluded, an NA
# value is applied)
graph %>%
  get_node_df()

# Create a new node attribute
# called `area`, which is the
# product of the `width` and
# `length` attributes
graph <-
  graph %>%
  mutate_node_attrs_ws(
    area = width * length)
```

```
# Get the graph's internal ndf
# to show that the node attribute
# values had been multiplied
# together (with new attr `area`)
# for nodes `2` and `3`
graph %>%
  get_node_df()

# We can invert the selection
# and mutate node `1` several
# times to get an `area` value
# for that node
graph <-
  graph %>%
  invert_selection() %>%
  mutate_node_attrs_ws(
    length = (log(width) + 5) %>%
               round(2),
    area = width * length)

# Get the graph's internal ndf
# to show that the 2 mutations
# occurred for node `1`, yielding
# non-NA values for its node
# attributes without changing
# those of the other nodes
graph %>%
  get_node_df()
```

---

node_aes *Insert node aesthetic attributes during node creation*

Description

This helper function should be invoked to provide values for the namesake node_aes argument, which is present in any function where nodes are created.

Usage

```
node_aes(shape = NULL, style = NULL, penwidth = NULL, color = NULL,
  fillcolor = NULL, fontname = NULL, fontsize = NULL, fontcolor = NULL,
  peripheries = NULL, height = NULL, width = NULL, x = NULL, y = NULL,
  group = NULL, tooltip = NULL, xlabel = NULL, URL = NULL,
  sides = NULL, orientation = NULL, skew = NULL, distortion = NULL,
  gradientangle = NULL, fixedsize = NULL, labelloc = NULL,
  margin = NULL)
```

Arguments

shape	the shape to use for the node. Some possible shape types include: circle, rectangle, triangle, plaintext, square, and polygon.
style	the node line style. The style types that can be used are filled, invisible, diagonals, rounded, dashed, dotted, solid, and bold.
penwidth	the thickness of the stroke line (in pt units) for the node shape. The default value is 1.0.
color	the color of the node's outline. Can be any of the named colors that R knows about (obtained using the colors() function), or, a hexadecimal color code.
fillcolor	the color with which to fill the shape of the node. Can be any of the named colors that R knows about (obtained using the colors() function), or, a hexadecimal color code.
fontname	the name of the system font that will be used for any node text.
fontsize	the point size of the font used for any node text.
fontcolor	the color used for any node text. Can be any of the named colors that R knows about (obtained using the colors() function), or, a hexadecimal color code.
peripheries	the repeated number of node shapes (of increasing size) to draw at the node perhipery.
height	the height of the node shape, in inches. The default value is 0.5 whereas the minimum value is 0.02. This is understood as the initial, minimum height of the node. If fixedsize is set to TRUE, this will be the final height of the node. Otherwise, if the node label requires more height to fit, the node's height will be increased to contain the label.
width	the width of the node shape, in inches. The default value is 0.5 whereas the minimum value is 0.02. This is understood as the initial, minimum width of the node. If fixedsize is set to TRUE, this will be the final width of the node. Otherwise, if the node label requires more width to fit, the node's width will be increased to contain the label.
x	the fixed position of the node in the x direction. Any integer-based or floating point value will be accepted.
y	the fixed position of the node in the y direction. Any integer-based or floating point value will be accepted.
group	the node group.
tooltip	text for a node tooltip.
xlabel	External label for a node. The label will be placed outside of the node but near it. These labels are added after all nodes and edges have been placed. The labels will be placed so that they do not overlap any node or label. This means it may not be possible to place all of them.
URL	a URL to associate with a node. Upon rendering the plot, clicking nodes with any associated URLs will open the URL in the default browser.
sides	when using the shape polygon, this value will provide the number of sides for that polygon.

orientation	this is the angle, in degrees, that is used to rotate nodes that have a shape of polygon. Not that for any of the polygon shapes (set by the sides node attribute), a value for orientation that is 0 results in a flat base.
skew	a 0-1 value that will result in the node shape being skewed to the right (from bottom to top). A value in the range 0 to -1 will skew the shape to the left.
distortion	a distortion factor that is used only when a shape of polygon is used. A 0-1 value will increasingly result in the top part of the node polygon shape to be larger than the bottom. Moving from 0 toward -1 will result in the opposite distortion effect.
gradientangle	the path angle for the node color fill gradient.
fixedsize	if set to FALSE, the size of a node is determined by smallest width and height needed to contain its label, if any, with a margin specified by the margin node attribute. The width and height must also be at least as large as the sizes specified by the width and height node attributes, which specify the minimum values. If set to TRUE, the node size is entirely specified by the values of the width and height node attributes (i.e., the node is not expanded in size to contain the text label).
labelloc	sets the vertical placement of labels for nodes and clusters. This attribute is used only when the height of the node is larger than the height of its label. The labelloc node attribute can be set to either t (top), c (center), or b (bottom). By default, the label is vertically centered.
margin	sets the amount of space around the node's label. By default, the value is 0.11,0.055.

Examples

```
# Create a new graph and add
# a path with several node
# aesthetic attributes
graph <-
  create_graph() %>%
  add_path(
    n = 3,
    type = "path",
    node_aes = node_aes(
      shape = "circle",
      x = c(1, 3, 2),
      y = c(4, -1, 3)))

# View the graph's internal
# node data frame; the node
# aesthetic attributes have
# been inserted
graph %>%
  get_node_df()
```

node_data *Insert node data attributes during node creation*

Description

This helper function should be invoked to provide values for the namesake `node_data` argument, which is present in any function where nodes are created.

Usage

```
node_data(...)
```

Arguments

`...`	node data attributes provided as one or more named vectors.

Examples

```
# Create a new graph and add
# a path with several node
# data attributes
graph <-
  create_graph() %>%
  add_path(
    n = 3,
    type = "path",
    node_data = node_data(
      hour = 5,
      index = c(1, 3, 2)))

# View the graph's internal
# node data frame; the node
# data attributes have been
# inserted
graph %>%
  get_node_df()
```

node_list_1 *Node list - Version 1.*

Description

A very simple, 2-column data frame that can be used to generate graph nodes.

Usage

```
node_list_1
```

Format

A data frame with 10 rows and 2 variables:

id a unique, monotonically increasing integer ID value

label a unique label associated with each ID value

---

node_list_2 *Node list - Version 2.*

Description

A simple, 5-column data frame that can be used to generate graph nodes.

Usage

```
node_list_2
```

Format

A data frame with 10 rows and 5 variables:

id a unique, monotonically increasing integer ID value

label a unique label associated with each ID value

type a grouping variable of either x, y, or z

value_1 a randomized set of numeric values between 0 and 10

value_2 a randomized set of numeric values between 0 and 10

---

nudge_node_positions_ws *Move layout positions of a selection of nodes*

Description

With an active selection of nodes, move the position in either the `x` or `y` directions, or both. Nodes in the selection that do not have position information (i.e., `NA` values for the `x` or `y` node attributes) will be ignored.

Usage

```
nudge_node_positions_ws(graph, dx, dy)
```

Arguments

graph	a graph object of class dgr_graph.
dx	a single numeric value specifying the amount that selected nodes (with non-NA values for the x and y attributes) will be moved in the x direction. A positive value will move nodes right, negative left.
dy	a single numeric value specifying the amount that selected nodes (with non-NA values for the x and y attributes) will be moved in the y direction. A positive value will move nodes up, negative down.

Value

a graph object of class dgr_graph.

Examples

```
# Create a simple graph with 4 nodes
graph <-
  create_graph() %>%
  add_node(
    type = "a",
    label = "one") %>%
  add_node(
    type = "a",
    label = "two") %>%
  add_node(
    type = "b",
    label = "three") %>%
  add_node(
    type = "b",
    label = "four")

# Add position information to each of
# the graph's nodes
graph <-
  graph %>%
  set_node_position(
    node = 1, x = 1, y = 1) %>%
  set_node_position(
    node = 2, x = 2, y = 2) %>%
  set_node_position(
    node = 3, x = 3, y = 3) %>%
  set_node_position(
    node = 4, x = 4, y = 4)

# Select all of the graph's nodes using the
# `select_nodes()` function (and only
# specifying the graph object)
graph <- select_nodes(graph)

# Move the selected nodes (all the nodes,
# in this case) 5 units to the right
```

```
graph <-
  graph %>%
  nudge_node_positions_ws(
    dx = 5, dy = 0)

# View the graph's node data frame
graph %>%
  get_node_df()

# Now select nodes that have `type == "b"`
# and move them in the `y` direction 2 units
# (the graph still has an active selection
# and so it must be cleared first)
graph <-
  graph %>%
  clear_selection() %>%
  select_nodes(
    conditions = type == "b") %>%
  nudge_node_positions_ws(
    dx = 0, dy = 2)

# View the graph's node data frame
graph %>%
  get_node_df()
```

---

open_graph *Read a graph or graph series from disk*

Description

Load a graph or a graph series object from disk.

Usage

```
open_graph(file)
```

Arguments

file	the filename for the graph or graph series. Optionally, this may contain a path to the file.

Examples

```
# Create an undirected GNP
# graph with 100 nodes using
# a probability value of 0.05
gnp_graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnp_graph(
```

```
    n = 100,
    p = 0.05)

# Save the graph to disk; use
# the file name `gnp_graph.dgr`
save_graph(
  x = gnp_graph,
  file = "gnp_graph")

# To read the graph file from
# disk, use `open_graph()`
gnp_graph_2 <-
  open_graph(
    file = "gnp_graph.dgr")
```

`recode_edge_attrs` *Recode a set of edge attribute values*

Description

Within a graph's internal edge data frame (edf), recode character or numeric edge attribute values. Optionally, one can specify a replacement value for any unmatched mappings.

Usage

```
recode_edge_attrs(graph, edge_attr_from, ..., otherwise = NULL,
  edge_attr_to = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`edge_attr_from`	the name of the edge attribute column from which values will be recoded.
`...`	single-length character vectors with the recoding instructions. The first component should have the value to replace and the second should have the replacement value (in the form `"[to_replace] -> [replacement]", ...`).
`otherwise`	an optional single value for recoding any unmatched values.
`edge_attr_to`	an optional name of a new edge attribute to which the recoded values will be applied. This will retain the original edge attribute and its values.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 4,
    m = 6,
    set_seed = 23) %>%
  set_edge_attrs(
    edge_attr = rel,
    values = c("a", "b", "a",
               "c", "b", "d"))

# Get the graph's internal edf
# to show which edge attributes
# are available
graph %>%
  get_edge_df()

# Recode the `rel` node
# attribute, creating a new edge
# attribute called `penwidth`;
# here, `a` is recoded to `1.0`,
# `b` maps to `1.5`, and all
# other values become `0.5`
graph <-
  graph %>%
  recode_edge_attrs(
    edge_attr_from = rel,
    "a -> 1.0",
    "b -> 1.5",
    otherwise = 0.5,
    edge_attr_to = penwidth)

# Get the graph's internal edf
# to show that the node
# attribute values had been
# recoded and copied into a
# new node attribute
graph %>%
  get_edge_df()
```

`recode_node_attrs` *Recode a set of node attribute values*

Description

Within a graph's internal node data frame (ndf), recode character or numeric node attribute values. Optionally, one can specify a replacement value for any unmatched mappings.

Usage

```
recode_node_attrs(graph, node_attr_from, ..., otherwise = NULL,
  node_attr_to = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
node_attr_from	the name of the node attribute column from which values will be recoded.
...	single-length character vectors with the recoding instructions. The first component should have the value to replace and the second should have the replacement value (in the form "[to_replace] -> [replacement]", ...).
otherwise	an optional single value for recoding any unmatched values.
node_attr_to	an optional name of a new node attribute to which the recoded values will be applied. This will retain the original node attribute and its values.

Value

a graph object of class dgr_graph.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 5,
    m = 10,
    set_seed = 23) %>%
  set_node_attrs(
    node_attr = shape,
    values =
      c("circle", "hexagon",
        "rectangle", "rectangle",
        "circle"))

# Get the graph's internal ndf
# to show which node
# attributes are available
graph %>%
  get_node_df()

# Recode the `shape` node
# attribute, so that `circle`
# is recoded to `square` and that
# `rectangle` becomes `triangle`
graph <-
  graph %>%
  recode_node_attrs(
    node_attr_from = shape,
```

```
    "circle -> square",
    "rectangle -> triangle")

# Get the graph's internal
# ndf to show that the node
# attribute values had been recoded
graph %>%
  get_node_df()

# Create a new node attribute,
# called `color`, that is based
# on a recoding of `shape`; here,
# map the square shape to a `red`
# color and map all other shapes
# to a `green` color
graph <-
  graph %>%
  recode_node_attrs(
    node_attr_from = shape,
    "square -> red",
    otherwise = "green",
    node_attr_to = color)

# Get the graph's internal ndf
# to see the change
graph %>%
  get_node_df()
```

`remove_graph_from_graph_series`

Remove a graph from a graph series

Description

Remove a single graph object from an set of graph objects contained within a graph series object.

Usage

```
remove_graph_from_graph_series(graph_series, index = "last")
```

Arguments

`graph_series`	a graph series object from which the graph object will be removed.
`index`	the index of the graph object to be removed from the graph series object.

Value

a graph series object of type `dgr_graph_1D`.

Examples

```
# Create three graphs
graph_1 <-
  create_graph() %>%
  add_path(n = 4)

graph_2 <-
  create_graph() %>%
  add_cycle(n = 5)

graph_3 <-
  create_graph() %>%
  add_star(n = 6)

# Create an empty graph series
# and add the graphs
series <-
  create_graph_series() %>%
  add_graph_to_graph_series(
    graph = graph_1) %>%
  add_graph_to_graph_series(
    graph = graph_2) %>%
  add_graph_to_graph_series(
    graph = graph_3)

# Remove the second graph
# from the graph series
series <-
  series %>%
  remove_graph_from_graph_series(
    index = 2)

# With `get_graph_series_info()`,
# we can ensure that a graph
# was removed
series %>%
  get_graph_series_info()
```

`rename_edge_attrs` *Rename an edge attribute*

Description

Within a graph's internal edge data frame (edf), rename an existing edge attribute.

Usage

```
rename_edge_attrs(graph, edge_attr_from, edge_attr_to)
```

Arguments

graph	a graph object of class dgr_graph.
edge_attr_from	the name of the edge attribute that will be renamed.
edge_attr_to	the new name of the edge attribute column identified in edge_attr_from.

Value

a graph object of class dgr_graph.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 5,
    m = 8,
    set_seed = 23) %>%
  set_edge_attrs(
    edge_attr = color,
    values = "green")

# Get the graph's internal edf
# to show which edge attributes
# are available
graph %>%
  get_edge_df()

# Rename the `color` node
# attribute as `weight`
graph <-
  graph %>%
  rename_edge_attrs(
    edge_attr_from = color,
    edge_attr_to = labelfontcolor)

# Get the graph's internal
# edf to show that the edge
# attribute had been renamed
graph %>%
  get_edge_df()
```

---

rename_node_attrs *Rename a node attribute*

Description

Within a graph's internal node data frame (ndf), rename an existing node attribute.

Usage

```
rename_node_attrs(graph, node_attr_from, node_attr_to)
```

Arguments

graph	a graph object of class dgr_graph.
node_attr_from	the name of the node attribute that will be renamed.
node_attr_to	the new name of the node attribute column identified in node_attr_from.

Value

a graph object of class dgr_graph.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 5,
    m = 8,
    set_seed = 23) %>%
  set_node_attrs(
    node_attr = shape,
    values = "circle") %>%
  set_node_attrs(
    node_attr = value,
    values = rnorm(
      n = count_nodes(.),
      mean = 5,
      sd = 1) %>% round(1))

# Get the graph's internal ndf
# to show which node attributes
# are available
graph %>%
  get_node_df()

# Rename the `value` node
# attribute as `weight`
graph <-
  graph %>%
  rename_node_attrs(
    node_attr_from = value,
    node_attr_to = weight)

# Get the graph's internal
# ndf to show that the node
# attribute had been renamed
graph %>%
```

```
get_node_df()
```

`renderDiagrammeR` *Widget render function for use in Shiny*

Description

Widget render function for use in Shiny

Usage

```
renderDiagrammeR(expr, env = parent.frame(), quoted = FALSE)
```

Arguments

`expr`	an expression that generates a DiagrammeR graph
`env`	the environment in which to evaluate expr.
`quoted`	is expr a quoted expression (with quote())? This is useful if you want to save an expression in a variable.

`renderGrViz` *Widget render function for use in Shiny*

Description

Widget render function for use in Shiny

Usage

```
renderGrViz(expr, env = parent.frame(), quoted = FALSE)
```

Arguments

`expr`	an expression that generates a DiagrammeR graph
`env`	the environment in which to evaluate expr.
`quoted`	is expr a quoted expression (with quote())? This is useful if you want to save an expression in a variable.

See Also

`grVizOutput` for an example in Shiny

render_graph *Render the graph in various formats*

Description

Using a dgr_graph object, render the graph in the RStudio Viewer.

Usage

```
render_graph(graph, layout = NULL, output = NULL, title = NULL,
  width = NULL, height = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
layout	a string specifying a layout type to use for node placement in this rendering. Possible layouts include: nicely, circle, tree, kk, and fr.
output	a string specifying the output type; graph (the default) renders the graph using the grViz function and visNetwork renders the graph using the visnetwork function.
title	an optional title for a graph when using output = "graph".
width	an optional parameter for specifying the width of the resulting graphic in pixels.
height	an optional parameter for specifying the height of the resulting graphic in pixels.

Examples

```
## Not run:
# Render a graph that's a
# balanced tree
create_graph() %>%
  add_balanced_tree(
    k = 2, h = 3) %>%
  render_graph()

# Use the `tree` layout for
# better node placement in this
# hierarchical graph
create_graph() %>%
  add_balanced_tree(
    k = 2, h = 3) %>%
  render_graph(layout = "tree")

# Plot the same tree graph but
# don't show the node ID values
create_graph() %>%
  add_balanced_tree(
    k = 2, h = 3) %>%
```

```
  set_node_attr_to_display() %>%
  render_graph(layout = "tree")

# Create a circle graph
create_graph() %>%
  add_gnm_graph(
    n = 55,
    m = 75,
    set_seed = 23) %>%
  render_graph(
    layout = "circle")

# Render the graph using the
# `visNetwork` output option
create_graph() %>%
  add_balanced_tree(
    k = 2, h = 3) %>%
  render_graph(
    output = "visNetwork")

## End(Not run)
```

`render_graph_from_graph_series` *Render a graph available in a series*

Description

Using a graph series object of type `dgr_graph_1D`, either render graph in the Viewer or output in various formats.

Usage

```
render_graph_from_graph_series(graph_series, graph_no, output = "graph",
  width = NULL, height = NULL)
```

Arguments

`graph_series`	a graph series object of type `dgr_graph_1D`.
`graph_no`	the index of the graph in the graph series.
`output`	a string specifying the output type; `graph` (the default) renders the graph using the `grViz` function, `DOT` outputs DOT code for the graph, and `SVG` provides SVG code for the rendered graph.
`width`	an optional parameter for specifying the width of the resulting graphic in pixels.
`height`	an optional parameter for specifying the height of the resulting graphic in pixels.

Examples

```
## Not run:
# Create three graphs
graph_1 <-
  create_graph() %>%
  add_path(n = 4)

graph_2 <-
  create_graph() %>%
  add_cycle(n = 5)

graph_3 <-
  create_graph() %>%
  add_star(n = 6)

# Create an empty graph series
# and add the graphs
series <-
  create_graph_series() %>%
  add_graph_to_graph_series(
    graph = graph_1) %>%
  add_graph_to_graph_series(
    graph = graph_2) %>%
  add_graph_to_graph_series(
    graph = graph_3)

# View the second graph in
# the series in the Viewer
render_graph_from_graph_series(
  graph_series = series,
  graph_no = 2)

## End(Not run)
```

---

reorder_graph_actions *Trigger the execution of a series of graph actions*

Description

Execute the graph actions stored in the graph through the use of the `add_graph_action()` function. These actions will be invoked in order and any errors encountered will trigger a warning message and result in no change to the input graph.

Usage

```
reorder_graph_actions(graph, indices)
```

Arguments

graph	a graph object of class dgr_graph.
indices	a numeric vector that provides the new ordering of graph actions. This vector can be the same length as the number of graph actions, or, of shorter length. In the latter case, the ordering places the given items first and the remaining actions will follow.

Value

a graph object of class dgr_graph.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 4,
    m = 4,
    set_seed = 23)

# Add three graph actions to the
# graph
graph <-
  graph %>%
  add_graph_action(
    fcn = "rescale_node_attrs",
    node_attr_from = "pagerank",
    node_attr_to = "width",
    action_name = "pgrnk_to_width") %>%
  add_graph_action(
    fcn = "set_node_attr_w_fcn",
    node_attr_fcn = "get_pagerank",
    column_name = "pagerank",
    action_name = "get_pagerank") %>%
  add_graph_action(
    fcn = "colorize_node_attrs",
    node_attr_from = "width",
    node_attr_to = "fillcolor",
    action_name = "pgrnk_fillcolor")

# View the graph actions for the graph
# object by using the function called
# `get_graph_actions()`
graph %>%
  get_graph_actions()

# We note that the order isn't
# correct and that the `get_pagerank`
# action should be the 1st action
```

```
# and `pgrnk_to_width` should go
# in 2nd place; to fix this, use the
# function `reorder_graph_actions()`
# and specify the reordering with a
# numeric vector
graph <-
  graph %>%
  reorder_graph_actions(
    indices = c(2, 1, 3))

# View the graph actions for the graph
# object once again to verify that
# we have the desired order of actions
graph %>%
  get_graph_actions()
```

`replace_in_spec` *Razor-like template for diagram specification*

Description

Use Razor-like syntax to define a template for use in a `grViz` diagram.

Usage

```
replace_in_spec(spec)
```

Arguments

`spec`	string spec to be parsed and evaluated

Examples

```
## Not run:
# a simple example to use a LETTER as a node label
spec <- "
  digraph { '@1' }

 [1]: LETTERS[1]
"
grViz(replace_in_spec(spec))


spec <- "
digraph a_nice_graph {
node [fontname = Helvetica]
a [label = '@1']
b [label = '@2-1']
c [label = '@2-2']
d [label = '@2-3']
```

```
e [label = '@2-4']
f [label = '@2-5']
g [label = '@2-6']
h [label = '@2-7']
i [label = '@2-8']
j [label = '@2-9']
a -> { b c d e f g h i j}
}

[1]: 'top'
[2]: 10:20
"
grViz(replace_in_spec(spec))

## End(Not run)
```

`rescale_edge_attrs` *Rescale numeric edge attribute values*

Description

From a graph object of class `dgr_graph`, take a set of numeric values for an edge attribute, rescale to a new numeric or color range, then write to the same edge attribute or to a new edge attribute column.

Usage

```
rescale_edge_attrs(graph, edge_attr_from, to_lower_bound = 0,
  to_upper_bound = 1, edge_attr_to = NULL, from_lower_bound = NULL,
  from_upper_bound = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`edge_attr_from`	the edge attribute containing numeric data that is to be rescaled to new numeric or color values.
`to_lower_bound`	the lower bound value for the set of rescaled values. This can be a numeric value or an X11 color name.
`to_upper_bound`	the upper bound value for the set of rescaled values. This can be a numeric value or an X11 color name.
`edge_attr_to`	an optional name of a new edge attribute to which the recoded values will be applied. This will retain the original edge attribute and its values.
`from_lower_bound`	an optional, manually set lower bound value for the rescaled values. If not set, the minimum value from the set will be used.
`from_upper_bound`	an optional, manually set upper bound value for the rescaled values. If not set, the minimum value from the set will be used.

Value

a graph object of class dgr_graph.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 10,
    m = 7,
    set_seed = 23) %>%
  set_edge_attrs(
    edge_attr = weight,
    values = rnorm(
      n = count_edges(.),
      mean = 5,
      sd = 1))

# Get the graph's internal edf
# to show which edge attributes
# are available
graph %>%
  get_edge_df()

# Rescale the `weight` edge
# attribute, so that its values
# are rescaled between 0 and 1
graph <-
  graph %>%
  rescale_edge_attrs(
    edge_attr_from = weight,
    to_lower_bound = 0,
    to_upper_bound = 1)

# Get the graph's internal edf
# to show that the edge attribute
# values had been rescaled
graph %>%
  get_edge_df()

# Scale the values in the `weight`
# edge attribute to different
# shades of gray for the `color`
# edge attribute and different
# numerical values for the
# `penwidth` attribute
graph <-
  graph %>%
  rescale_edge_attrs(
    edge_attr_from = weight,
```

```
    to_lower_bound = "gray80",
    to_upper_bound = "gray20",
    edge_attr_to = color) %>%
  rescale_edge_attrs(
    edge_attr_from = weight,
    to_lower_bound = 0.5,
    to_upper_bound = 3,
    edge_attr_to = penwidth)

# Get the graph's internal edf
# once more to show that scaled
# grayscale colors are now available
# in `color` and scaled numerical
# values are in the `penwidth`
# edge attribute
graph %>%
  get_edge_df()
```

`rescale_node_attrs` *Rescale numeric node attribute values*

Description

From a graph object of class `dgr_graph`, take a set of numeric values for a node attribute, rescale to a new numeric or color range, then write to the same node attribute or to a new node attribute column.

Usage

```
rescale_node_attrs(graph, node_attr_from, to_lower_bound = 0,
  to_upper_bound = 1, node_attr_to = NULL, from_lower_bound = NULL,
  from_upper_bound = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`node_attr_from`	the node attribute containing numeric data that is to be rescaled to new numeric or color values.
`to_lower_bound`	the lower bound value for the set of rescaled values. This can be a numeric value or an X11 color name.
`to_upper_bound`	the upper bound value for the set of rescaled values. This can be a numeric value or an X11 color name.
`node_attr_to`	an optional name of a new node attribute to which the recoded values will be applied. This will retain the original node attribute and its values.
`from_lower_bound`	an optional, manually set lower bound value for the rescaled values. If not set, the minimum value from the set will be used.

from_upper_bound
: an optional, manually set upper bound value for the rescaled values. If not set, the minimum value from the set will be used.

Value

a graph object of class dgr_graph.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 5,
    m = 10,
    set_seed = 23) %>%
  set_node_attrs(
    node_attr = value,
    values = rnorm(
      n = count_nodes(.),
      mean = 5,
      sd = 1) %>% round(1))

# Get the graph's internal ndf
# to show which node attributes
# are available
graph %>%
  get_node_df()

# Rescale the `value` node
# attribute, so that its values
# are rescaled between 0 and 1
graph <-
  graph %>%
  rescale_node_attrs(
    node_attr_from = value,
    to_lower_bound = 0,
    to_upper_bound = 1)

# Get the graph's internal ndf
# to show that the node attribute
# values had been rescaled
graph %>%
  get_node_df()

# Scale the values in the `value`
# node attribute to different
# shades of gray for the `fillcolor`
# and `fontcolor` node attributes
graph <-
  graph %>%
```

```
  rescale_node_attrs(
    node_attr_from = value,
    to_lower_bound = "gray80",
    to_upper_bound = "gray20",
    node_attr_to = fillcolor) %>%
  rescale_node_attrs(
    node_attr_from = value,
    to_lower_bound = "gray5",
    to_upper_bound = "gray95",
    node_attr_to = fontcolor)

# Get the graph's internal ndf
# once more to show that scaled
# grayscale colors are now available
# in the `fillcolor` and `fontcolor`
# node attributes
graph %>%
  get_node_df()
```

`rev_edge_dir` *Reverse the direction of all edges in a graph*

Description

Using a directed graph as input, reverse the direction of all edges in that graph.

Usage

```
rev_edge_dir(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a graph with a
# directed tree
graph <-
  create_graph() %>%
  add_balanced_tree(
    k = 2, h = 2)

# Inspect the graph's edges
graph %>%
  get_edges()
```

```
# Reverse the edge directions
# such that edges are directed
# toward the root of the tree
graph <-
  graph %>%
  rev_edge_dir()

# Inspect the graph's edges
# after their reversal
graph %>%
  get_edges()
```

`rev_edge_dir_ws` *Reverse the direction of selected edges in a graph*

Description

Using a directed graph with a selection of edges as input, reverse the direction of those selected edges in input graph.

Usage

```
rev_edge_dir_ws(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a graph with a
# directed tree
graph <-
  create_graph() %>%
  add_balanced_tree(
    k = 2, h = 2)

# Inspect the graph's edges
graph %>%
  get_edges()

# Select all edges associated
# with nodes `1` and `2`
graph <-
  graph %>%
```

```
  select_edges_by_node_id(
    nodes = 1:2)

# Reverse the edge directions
# of the edges associated with
# nodes `1` and `2`
graph <-
  graph %>%
  rev_edge_dir_ws()

# Inspect the graph's edges
# after their reversal
graph %>%
  get_edges()
```

---

save_graph *Save a graph or graph series to disk*

Description

Save a graph or a graph series object to disk.

Usage

```
save_graph(x, file)
```

Arguments

x	a graph object of class dgr_graph or a graph series object of type dgr_graph_1D.
file	a file name for the graph or graph series. Provide a character string and the .dgr extension will be applied to it.

Examples

```
# Create an undirected GNP
# graph with 100 nodes using
# a probability value of 0.05
gnp_graph <-
  create_graph(
    directed = FALSE) %>%
  add_gnp_graph(
    n = 100,
    p = 0.05)

# Save the graph to disk; use
# the file name `gnp_graph.dgr`
save_graph(
  x = gnp_graph,
  file = "gnp_graph")
```

```
# To read the graph file from
# disk, use `open_graph()`
gnp_graph_2 <-
  open_graph(
    file = "gnp_graph.dgr")
```

`select_edges` *Select edges in a graph*

Description

Select edges from a graph object of class `dgr_graph`.

Usage

```
select_edges(graph, conditions = NULL, set_op = "union", from = NULL,
  to = NULL, edges = NULL)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`conditions`	an option to use filtering conditions for the retrieval of edges.
`set_op`	the set operation to perform upon consecutive selections of graph nodes. This can either be as a `union` (the default), as an intersection of selections with `intersect`, or, as a `difference` on the previous selection, if it exists.
`from`	an optional vector of node IDs from which the edge is outgoing for filtering the list of edges present in the graph.
`to`	an optional vector of node IDs to which the edge is incoming for filtering the list of edges present in the graph.
`edges`	an optional vector of edge IDs for filtering the list of edges present in the graph.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a node data frame (ndf)
ndf <-
  create_node_df(
    n = 4,
    type = "basic",
    label = TRUE,
    value = c(3.5, 2.6, 9.4, 2.7))

# Create an edge data frame (edf)
```

```
edf <-
  create_edge_df(
    from = c(1, 2, 3),
    to = c(4, 3, 1),
    rel = c("a", "z", "a"),
    value = c(6.4, 2.9, 5.0))

# Create a graph with the ndf and edf
graph <-
  create_graph(
    nodes_df = ndf,
    edges_df = edf)

# Explicitly select the edge `1`->`4`
graph <-
  graph %>%
  select_edges(
    from = 1,
    to = 4)

# Verify that an edge selection has been made
# using the `get_selection()` function
graph %>%
  get_selection()

# Select edges based on the relationship label
# being `z`
graph <-
  graph %>%
  clear_selection() %>%
  select_edges(
    conditions = rel == "z")

# Verify that an edge selection has been made, and
# recall that the `2`->`3` edge uniquely has the
# `z` relationship label
graph %>%
  get_selection()

# Select edges based on the edge value attribute
# being greater than 3.0 (first clearing the current
# selection of edges)
graph <-
  graph %>%
  clear_selection() %>%
  select_edges(
    conditions = value > 3.0)

# Verify that the correct edge selection has been
# made; in this case, edges `1`->`4` and
# `3`->`1` have values for `value` > 3.0
graph %>%
  get_selection()
```

`select_edges_by_edge_id` *Select edges in a graph using edge ID values*

Description

Select edges in a graph object of class `dgr_graph` using edge ID values.

Usage

```
select_edges_by_edge_id(graph, edges, set_op = "union")
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`edges`	a vector of edge IDs for the selection of edges present in the graph.
`set_op`	the set operation to perform upon consecutive selections of graph edges This can either be as a `union` (the default), as an intersection of selections with `intersect`, or, as a `difference` on the previous selection, if it exists.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a graph with 5 nodes
graph <-
  create_graph() %>%
  add_path(n = 5)

# Create a graph selection by selecting
# edges with edge IDs `1` and `2`
graph <-
  graph %>%
  select_edges_by_edge_id(
    edges = 1:2)

# Get the selection of edges
graph %>%
  get_selection()

# Perform another selection of edges,
# with edge IDs `1`, `2`, and `4`
graph <-
  graph %>%
  clear_selection() %>%
  select_edges_by_edge_id(
```

```
    edges = c(1, 2, 4))

# Get the selection of edges
graph %>%
  get_selection()

# Get the fraction of edges selected
# over all the edges in the graph
graph %>%
  {
    l <- get_selection(.) %>%
      length(.)
    e <- count_edges(.)
    l/e
  }
```

`select_edges_by_node_id` *Select edges in a graph using node ID values*

Description

Select edges in a graph object of class `dgr_graph` using node ID values. All edges associated with the provided nodes will be included in the selection.

Usage

```
select_edges_by_node_id(graph, nodes, set_op = "union")
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`nodes`	a vector of node IDs for the selection of edges present in the graph.
`set_op`	the set operation to perform upon consecutive selections of graph edges This can either be as a `union` (the default), as an intersection of selections with `intersect`, or, as a `difference` on the previous selection, if it exists.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a graph with 5 nodes
graph <-
  create_graph() %>%
  add_path(n = 5)

# Create a graph selection by selecting edges
```

```
# associated with nodes `1` and `2`
graph <-
  graph %>%
  select_edges_by_node_id(
    nodes = 1:2)

# Get the selection of edges
graph %>%
  get_selection()

# Perform another selection of edges, with nodes
# `1`, `2`, and `4`
graph <-
  graph %>%
  clear_selection() %>%
  select_edges_by_node_id(
    nodes = c(1, 2, 4))

# Get the selection of edges
graph %>%
  get_selection()

# Get a fraction of the edges selected over all
# the edges in the graph
graph %>%
  {
    l <- get_selection(.) %>%
      length(.)
    e <- count_edges(.)
    l/e
  }
```

`select_last_edges_created` *Select the last set of edges created in a graph*

Description

Select the last edges that were created in a graph object of class `dgr_graph`. This function should ideally be used just after creating the edges to be selected.

Usage

```
select_last_edges_created(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a graph object of class dgr_graph.

Examples

```
# Create a graph and add a cycle and then
# a tree in 2 separate function calls
graph <-
  create_graph() %>%
  add_cycle(
    n = 3,
    rel = "a") %>%
  add_balanced_tree(
    k = 2, h = 2,
    rel = "b")

# Select the last edges created (all edges
# from the tree) and then set their edge
# color to be `red`
graph <-
  graph %>%
  select_last_edges_created() %>%
  set_edge_attrs_ws(
    edge_attr = color,
    value = "red") %>%
  clear_selection()

# Display the graph's internal edge
# data frame to verify the change
graph %>%
  get_edge_df()
```

---

select_last_nodes_created

Select the last set of nodes created in a graph

Description

Select the last nodes that were created in a graph object of class dgr_graph. This function should ideally be used just after creating the nodes to be selected.

Usage

```
select_last_nodes_created(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a graph object of class dgr_graph.

Examples

```
# Create a graph and add 4 nodes
# in 2 separate function calls
graph <-
  create_graph() %>%
  add_n_nodes(
    n = 2,
    type = "a",
    label = c("a_1", "a_2")) %>%
  add_n_nodes(
    n = 2,
    type = "b",
    label = c("b_1", "b_2"))

# Select the last nodes created (2 nodes
# from the last function call) and then
# set their color to be `red`
graph <-
  graph %>%
  select_last_nodes_created() %>%
  set_node_attrs_ws(
    node_attr = color,
    value = "red") %>%
  clear_selection()

# Display the graph's internal node
# data frame to verify the change
graph %>%
  get_node_df()
```

---

select_nodes *Select nodes in a graph*

Description

Select nodes from a graph object of class dgr_graph.

Usage

```
select_nodes(graph, conditions = NULL, set_op = "union", nodes = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
conditions	an option to use filtering conditions for the retrieval of nodes.
set_op	the set operation to perform upon consecutive selections of graph nodes. This can either be as a union (the default), as an intersection of selections with intersect, or, as a difference on the previous selection, if it exists.
nodes	an optional vector of node IDs for filtering the list of nodes present in the graph.

Value

a graph object of class dgr_graph.

Examples

```
# Create a node data frame (ndf)
ndf <-
  create_node_df(
    n = 4,
    type = c("a", "a", "z", "z"),
    label = TRUE,
    value = c(3.5, 2.6, 9.4, 2.7))

# Create an edge data frame (edf)
edf <-
  create_edge_df(
    from = c(1, 2, 3),
    to = c(4, 3, 1),
    rel = c("a", "z", "a"))

# Create a graph with the ndf and edf
graph <-
  create_graph(
    nodes_df = ndf,
    edges_df = edf)

# Explicitly select nodes `1` and `3`
graph <-
  graph %>%
  select_nodes(nodes = c(1, 3))

# Verify that the node selection has been made
# using the `get_selection()` function
graph %>%
  get_selection()

# Select nodes based on the node `type`
# being `z`
graph <-
  graph %>%
  clear_selection() %>%
  select_nodes(
```

```
    conditions = type == "z")

# Verify that an node selection has been made, and
# recall that the `3` and `4` nodes are of the
# `z` type
graph %>%
  get_selection()

# Select edges based on the node value attribute
# being greater than 3.0 (first clearing the current
# selection of nodes)
graph <-
  graph %>%
  clear_selection() %>%
  select_nodes(
    conditions = value > 3.0)

# Verify that the correct node selection has been
# made; in this case, nodes `1` and `3` have values
# for `value` greater than 3.0
graph %>%
  get_selection()
```

`select_nodes_by_degree` *Select nodes in the graph based on their degree values*

Description

Using a graph object of class `dgr_graph`, create a selection of nodes that have certain degree values.

Usage

```
select_nodes_by_degree(graph, expressions, set_op = "union")
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`expressions`	one or more expressions for filtering nodes by degree values. Use a combination of degree type (`deg` for total degree, `indeg` for in-degree, and `outdeg` for out-degree) with a comparison operator and values for comparison (e.g., use `"deg >= 2"` to select nodes with a degree greater than or equal to 2).
`set_op`	the set operation to perform upon consecutive selections of graph nodes. This can either be as a `union` (the default), as an intersection of selections with `intersect`, or, as a `difference` on the previous selection, if it exists.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a random graph using
# the `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 35, m = 125,
    set_seed = 23)

# Report which nodes have a
# total degree (in-degree +
# out-degree) of exactly 9
graph %>%
  select_nodes_by_degree(
    expressions = "deg == 9") %>%
  get_selection()

# Report which nodes have a
# total degree greater than or
# equal to 9
graph %>%
  select_nodes_by_degree(
    expressions = "deg >= 9") %>%
  get_selection()

# Combine two calls of
# `select_nodes_by_degree()` to
# get those nodes with total
# degree less than 3 and total
# degree greater than 10 (by
# default, those `select...()`
# functions will `union` the
# sets of nodes selected)
graph %>%
  select_nodes_by_degree(
    expressions = "deg < 3") %>%
  select_nodes_by_degree(
    expressions = "deg > 10") %>%
  get_selection()

# Combine two calls of
# `select_nodes_by_degree()` to
# get those nodes with total
# degree greater than or equal
# to 3 and less than or equal
# to 10 (the key here is to
# `intersect` the sets of nodes
# selected in the second call)
graph %>%
  select_nodes_by_degree(
    expressions = "deg >= 3") %>%
  select_nodes_by_degree(
```

```
    expressions = "deg <= 10",
    set_op = "intersect") %>%
  get_selection()

# Select all nodes with an
# in-degree greater than 5, then,
# apply a node attribute to those
# selected nodes (coloring the
# selected nodes red)
graph_2 <-
  graph %>%
  select_nodes_by_degree(
    expressions = "indeg > 5") %>%
  set_node_attrs_ws(
    node_attr = color,
    value = "red")

# Get the selection of nodes
graph_2 %>%
  get_selection()
```

`select_nodes_by_id` *Select nodes in a graph by ID values*

Description

Select nodes in a graph object of class `dgr_graph` by their node ID values. If nodes have IDs that are monotonically increasing integer values, then numeric ranges can be used for the selection.

Usage

```
select_nodes_by_id(graph, nodes, set_op = "union")
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`nodes`	a vector of node IDs for the selection of nodes present in the graph.
`set_op`	the set operation to perform upon consecutive selections of graph nodes. This can either be as a `union` (the default), as an intersection of selections with `intersect`, or, as a `difference` on the previous selection, if it exists.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a node data frame (ndf)
ndf <-
  create_node_df(n = 10)

# Create a graph
graph <-
  create_graph(
    nodes_df = ndf)

# Select nodes `1` to `5` and show that
# selection of nodes with `get_selection()`
graph %>%
  select_nodes_by_id(nodes = 1:5) %>%
  get_selection()
```

---

select_nodes_in_neighborhood

Select nodes based on a walk distance from a specified node

Description

Select those nodes in the neighborhood of nodes connected a specified distance from an initial node.

Usage

```
select_nodes_in_neighborhood(graph, node, distance, set_op = "union")
```

Arguments

graph	a graph object of class dgr_graph.
node	the node from which the traversal will originate.
distance	the maximum number of steps from the node for inclusion in the selection.
set_op	the set operation to perform upon consecutive selections of graph nodes. This can either be as a union (the default), as an intersection of selections with intersect, or, as a difference on the previous selection, if it exists.

Value

a graph object of class dgr_graph.

Examples

```
# Create a graph containing
# a balanced tree
graph <-
  create_graph() %>%
  add_balanced_tree(
    k = 2, h = 2)

# Create a graph selection by
# selecting nodes in the
# neighborhood of node `1`, where
# the neighborhood is limited by
# nodes that are 1 connection
# away from node `1`
graph <-
  graph %>%
  select_nodes_in_neighborhood(
    node = 1,
    distance = 1)

# Get the selection of nodes
graph %>%
  get_selection()

# Perform another selection
# of nodes, this time with a
# neighborhood spanning 2 nodes
# from node `1`
graph <-
  graph %>%
  clear_selection() %>%
  select_nodes_in_neighborhood(
    node = 1,
    distance = 2)

# Get the selection of nodes
graph %>%
  get_selection()
```

---

set_cache *Cache a vector in the graph*

Description

Place any vector in the cache of a graph object of class dgr_graph.

Usage

```
set_cache(graph, to_cache, name = NULL, col = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
to_cache	any vector or data frame. If a data frame is supplied then a single column for the vector to pull must be provided in the col argument.
name	an optional name for the cached vector.
col	if a data frame is provided in to_cache then a column name from that data frame must provided here.

Value

a graph object of class dgr_graph.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 10,
    m = 22,
    set_seed = 23)

# Get the closeness values for
# all nodes from `1` to `10` and
# store in the graph's cache
graph <-
  graph %>%
  set_cache(
    name = "closeness_vector",
    to_cache = get_closeness(.),
    col = "closeness")

# Get the graph's cache
graph %>%
  get_cache(
    name = "closeness_vector")

# Get the difference of betweenness
# and closeness values for nodes in
# the graph and store the vector in
# the graph's cache
graph <-
  graph %>%
  set_cache(
    name = "difference",
    to_cache =
      get_betweenness(.)$betweenness -
        get_closeness(.)$closeness)

# Get the graph's cache
```

```
graph %>%
  get_cache(
    name = "difference")
```

`set_df_as_edge_attr` *Set a data frame as an edge attribute*

Description

From a graph object of class `dgr_graph`, bind a data frame as an edge attribute property for one given graph edge. The data frames are stored in list columns within a `df_tbl` object, itself residing within the graph object. A `df_id` value is generated and serves as a pointer to the table row that contains the ingested data frame.

Usage

```
set_df_as_edge_attr(graph, edge, df)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`edge`	the edge ID to which the data frame will be bound as an attribute.
`df`	the data frame to be bound to the edge as an attribute.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a node data frame (ndf)
ndf <-
  create_node_df(
    n = 4,
    type = "basic",
    label = TRUE,
    value = c(3.5, 2.6, 9.4, 2.7))

# Create an edge data frame (edf)
edf <-
  create_edge_df(
    from = c(1, 2, 3),
    to = c(4, 3, 1),
    rel = "leading_to")

# Create a graph
graph <-
  create_graph(
    nodes_df = ndf,
```

```
    edges_df = edf)

# Create a simple data frame to add as
# an edge attribute
df <-
  data.frame(
    a = c("one", "two", "three"),
    b = c(1, 2, 3),
    stringsAsFactors = FALSE)

# Bind the data frame as an edge attribute
# to the edge with ID `1`
graph <-
  graph %>%
  set_df_as_edge_attr(
    edge = 1,
    df = df)
```

---

set_df_as_node_attr *Set a data frame as a node attribute*

Description

From a graph object of class dgr_graph, bind a data frame as a node attribute property for one given graph node. The data frames are stored in list columns within a df_tbl object. A df_id value is generated and serves as a pointer to the table row that contains the ingested data frame.

Usage

```
set_df_as_node_attr(graph, node, df)
```

Arguments

graph	a graph object of class dgr_graph.
node	the node ID to which the data frame will be bound as an attribute.
df	the data frame to be bound to the node as an attribute.

Value

a graph object of class dgr_graph.

Examples

```
# Create a node data frame (ndf)
ndf <-
  create_node_df(
    n = 4,
    type = "basic",
    label = TRUE,
```

```
    value = c(3.5, 2.6, 9.4, 2.7))

# Create an edge data frame (edf)
edf <-
  create_edge_df(
    from = c(1, 2, 3),
    to = c(4, 3, 1),
    rel = "leading_to")

# Create a graph
graph <-
  create_graph(
    nodes_df = ndf,
    edges_df = edf)

# Create a simple data frame to add as
# a node attribute
df <-
  data.frame(
    a = c("one", "two", "three"),
    b = c(1, 2, 3),
    stringsAsFactors = FALSE)

# Bind the data frame as a node attribute
# of node `1`
graph <-
  graph %>%
  set_df_as_node_attr(
    node = 1,
    df = df)

# Create another data frame to add as
# a node attribute
df_2 <-
  data.frame(
    c = c("four", "five", "six"),
    d = c(4, 5, 6),
    stringsAsFactors = FALSE)

# Bind the data frame as a node attribute
# of node `2`
graph <-
  graph %>%
  set_df_as_node_attr(
    node = 2,
    df = df_2)
```

---

set_edge_attrs *Set edge attribute values*

Description

From a graph object of class dgr_graph, set edge attribute values for one or more edges.

Usage

```
set_edge_attrs(graph, edge_attr, values, from = NULL, to = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
edge_attr	the name of the attribute to set.
values	the values to be set for the chosen attribute for the chosen edges.
from	an optional vector of node IDs from which the edge is outgoing for filtering list of nodes with outgoing edges in the graph.
to	an optional vector of node IDs from which the edge is incoming for filtering list of nodes with incoming edges in the graph.

Value

a graph object of class dgr_graph.

Examples

```
# Create a simple graph
ndf <-
  create_node_df(
    n = 4,
    type = "basic",
    label = TRUE,
    value = c(3.5, 2.6, 9.4, 2.7))

edf <-
  create_edge_df(
    from = c(1, 2, 3),
    to = c(4, 3, 1),
    rel = "leading_to")

graph <-
  create_graph(
    nodes_df = ndf,
    edges_df = edf)

# Set attribute `color = "green"`
# for edges `1`->`4` and `3`->`1`
# in the graph
graph <-
  graph %>%
  set_edge_attrs(
    edge_attr = color,
    values = "green",
```

```
    from = c(1, 3),
    to = c(4, 1))

# Set attribute `color = "blue"`
# for all edges in the graph
graph <-
  graph %>%
  set_edge_attrs(
    edge_attr = color,
    values = "blue")

# Set attribute `color = "pink"`
# for all edges in graph outbound
# from node with ID value `1`
graph <-
  graph %>%
  set_edge_attrs(
    edge_attr = color,
    values = "pink",
    from = 1)

# Set attribute `color = "black"`
# for all edges in graph inbound
# to node with ID `1`
graph <-
  graph %>%
  set_edge_attrs(
    edge_attr = color,
    values = "black",
    to = 1)
```

`set_edge_attrs_ws` *Set edge attributes with an edge selection*

Description

From a graph object of class `dgr_graph` or an edge data frame, set edge attribute properties for one or more edges.

Selections of edges can be performed using the following `select_...` functions: `select_edges()`, `select_last_edge()`, or `select_edges_by_node_id()`. Selections of edges can also be performed using the following traversal functions: `trav_out_edge()`, `trav_in_edge()`, or `trav_both_edge()`.

Usage

```
set_edge_attrs_ws(graph, edge_attr, value)
```

Arguments

graph	a graph object of class dgr_graph.
edge_attr	the name of the attribute to set.
value	the value to be set for the chosen attribute for the edges in the current selection.

Value

a graph object of class dgr_graph.

Examples

```
# Create a simple graph
graph <-
  create_graph() %>%
  add_path(n = 6)

# Select specific edges from
# the graph and apply the edge
# attribute `color = blue` to
# those selected edges
graph <-
  graph %>%
  select_nodes_by_id(nodes = 2:4) %>%
  trav_out_edge() %>%
  set_edge_attrs_ws(
    edge_attr = color,
    value = "blue")

# Show the internal edge data
# frame to verify that the
# edge attribute has been set
# for specific edges
graph %>%
  get_edge_df()
```

---

set_edge_attr_to_display

Set the edge attribute values to be rendered

Description

Set a edge attribute type to display as edge text when calling the render_graph() function. This allows for display of different types of edge attribute values on a per-edge basis. Without setting the display attribute, rendering a graph will default to not printing any text on edges. Setting the display edge attribute with this function for the first time (i.e., the display column doesn't exist in the graph's internal edge data frame) will insert the attr value for all edges specified in edges and a default value (default) for all remaining edges.

Usage

```
set_edge_attr_to_display(graph, attr = NULL, edges = NULL,
  default = "label")
```

Arguments

graph	a graph object of class dgr_graph.
attr	the name of the attribute from which label text for the edge will be obtained. If set to NULL, then NA values will be assigned to the display column for the chosen edges.
edges	a length vector containing one or several edge ID values (as integers) for which edge attributes are set for display in the rendered graph. If NULL, all edges from the graph are assigned the display value given as attr.
default	the name of an attribute to set for all other graph edges not included in edges. This value only gets used if the display edge attribute is not in the graph's internal edge data frame.

Value

a graph object of class dgr_graph.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 4,
    m = 4,
    set_seed = 23) %>%
  set_edge_attrs(
    edge_attr = value,
    values = c(2.5, 8.2, 4.2, 2.4))

# For edge ID values of `1`,
# `2`, and `3`, choose to display
# the edge `value` attribute (for
# the other edges, display nothing)
graph <-
  graph %>%
  set_edge_attr_to_display(
    edges = 1:3,
    attr = value,
    default = NA)

# Show the graph's edge data frame; the
# `display` edge attribute will show, for
# each row, which edge attribute value to
# display when the graph is rendered
```

```
graph %>%
  get_edge_df()

# This function can be called multiple
# times on a graph; after the first time
# (i.e., creation of the `display`
# attribute), the `default` value won't
# be used
graph %>%
  set_edge_attr_to_display(
    edges = 4,
    attr = to) %>%
  set_edge_attr_to_display(
    edges = c(1, 3),
    attr = id) %>%
  get_edge_df()
```

`set_graph_directed` *Convert an undirected graph to a directed graph*

Description

Take a graph which is undirected and convert it to a directed graph.

Usage

```
set_graph_directed(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a graph with a
# undirected tree
graph <-
  create_graph(
    directed = FALSE) %>%
  add_balanced_tree(
    k = 2, h = 2)

# Convert this graph from
# undirected to directed
graph <-
  graph %>%
```

```
  set_graph_directed()

# Perform a check on whether
# graph is directed
graph %>%
  is_graph_directed()
```

`set_graph_name` *Set graph name*

Description

Set a name for a graph object of class `dgr_graph`.

Usage

```
set_graph_name(graph, name)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`name`	the name to set for the graph.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create an empty graph
graph <- create_graph()

# Provide the new graph with a name
graph <-
  graph %>%
  set_graph_name(
    name = "example_name")
```

set_graph_time *Set graph date-time and timezone*

Description

Set the time and timezone for a graph object of class dgr_graph.

Usage

```
set_graph_time(graph, time = NULL, tz = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
time	the date-time to set for the graph.
tz	the timezone to set for the graph.

Value

a graph object of class dgr_graph.

Examples

```
# Create an empty graph
graph <- create_graph()

# Provide the new graph with a timestamp (if `tz`
# is not supplied, `GMT` is used as the time zone)
graph_1 <-
  graph %>%
  set_graph_time(
    time = "2015-10-25 15:23:00")

# Provide the new graph with a timestamp that is
# the current time; the time zone is inferred from
# the user's locale
graph_2 <-
  graph %>%
  set_graph_time()

# The time zone can be updated when a timestamp
# is present
graph_2 <-
  graph_2 %>%
  set_graph_time(
    tz = "America/Los_Angeles")
```

set_graph_undirected *Convert a directed graph to an undirected graph*

Description

Take a graph which is directed and convert it to an undirected graph.

Usage

```
set_graph_undirected(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

a graph object of class dgr_graph.

Examples

```
# Create a graph with a
# directed tree
graph <-
  create_graph() %>%
  add_balanced_tree(
    k = 2, h = 2)

# Convert this graph from
# directed to undirected
graph <-
  graph %>%
  set_graph_undirected()

# Perform a check on whether
# graph is directed
graph %>%
  is_graph_directed()
```

set_node_attrs *Set node attribute values*

Description

From a graph object of class dgr_graph, set node attribute values for one or more nodes.

Usage

```
set_node_attrs(graph, node_attr, values, nodes = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
node_attr	the name of the attribute to set.
values	the values to be set for the chosen attribute for the chosen nodes.
nodes	an optional vector of node IDs for filtering the list of nodes present in the graph.

Value

a graph object of class dgr_graph.

Examples

```
# Create a node data frame (ndf)
ndf <-
  create_node_df(
    n = 4,
    type = "basic",
    label = TRUE,
    value = c(3.5, 2.6, 9.4, 2.7))

# Create an edge data frame (edf)
edf <-
  create_edge_df(
    from = c(1, 2, 3),
    to = c(4, 3, 1),
    rel = "leading_to")

# Create a graph
graph <-
  create_graph(
    nodes_df = ndf,
    edges_df = edf)

# Set attribute `color = "green"` for
# nodes `1` and `3` using the graph object
graph <-
  graph %>%
  set_node_attrs(
    node_attr = color,
    values = "green",
    nodes = c(1, 3))

# View the graph's node data frame
graph %>%
  get_node_df()

# Set attribute `color = "blue"` for
```

```
# all nodes in the graph
graph <-
  graph %>%
  set_node_attrs(
    node_attr = color,
    values = "blue")

# Display the graph's ndf
graph %>%
  get_node_df()
```

`set_node_attrs_ws` *Set node attributes with a node selection*

Description

From a graph object of class `dgr_graph` or a node data frame, set node attribute properties for nodes present in a node selection.

Selections of nodes can be performed using the following `select_...` functions: `select_nodes()`, `select_last_nodes_created()`, `select_nodes_by_degree()`, `select_nodes_by_id()`, or `select_nodes_in_neighbo`
Selections of nodes can also be performed using the following traversal functions: (`trav_...`): `trav_out()`, `trav_in()`, `trav_both()`, `trav_in_node()`, `trav_out_node()`.

Usage

```
set_node_attrs_ws(graph, node_attr, value)
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`node_attr`	the name of the attribute to set.
`value`	the value to be set for the chosen attribute for the nodes in the current selection.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a simple graph
graph <-
  create_graph() %>%
  add_path(n = 6)

# Select specific nodes from the graph and
# apply the node attribute `color = blue` to
# those selected nodes
graph <-
```

```
  graph %>%
  select_nodes_by_id(
    nodes = 1:4) %>%
  trav_out() %>%
  set_node_attrs_ws(
    node_attr = color,
    value = "blue")

# Show the internal node data frame to verify
# that the node attribute has been set for
# specific node
graph %>%
  get_node_df()
```

`set_node_attr_to_display` *Set the node attribute values to be rendered*

Description

Set a node attribute type to display as node text when calling the `render_graph()` function. This allows for display of different types of node attribute values on a per-node basis. Without setting the `display` attribute, rendering a graph will default to printing text from the `label` attribute on nodes. Setting the `display` node attribute with this function for the first time (i.e., the `display` column doesn't exist in the graph's internal node data frame) will insert the `attr` value for all nodes specified in `nodes` and a default value (`default`) for all remaining nodes.

Usage

```
set_node_attr_to_display(graph, attr = NULL, nodes = NULL,
  default = "label")
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`attr`	the name of the attribute from which label text for the node will be obtained. If set to `NULL`, then `NA` values will be assigned to the `display` column for the chosen nodes.
`nodes`	a length vector containing one or several node ID values (as integers) for which node attributes are set for display in the rendered graph. If `NULL`, all nodes from the graph are assigned the `display` value given as `attr`.
`default`	the name of an attribute to set for all other graph nodes not included in `nodes`. This value only gets used if the `display` node attribute is not in the graph's internal node data frame.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 4,
    m = 4,
    set_seed = 23) %>%
  set_node_attrs(
    node_attr = value,
    values = c(2.5, 8.2, 4.2, 2.4))

# For node ID values of `1`,
# `2`, and `3`, choose to display
# the node `value` attribute (for
# the other nodes, display nothing)
graph <-
  graph %>%
  set_node_attr_to_display(
    nodes = 1:3,
    attr = value,
    default = NA)

# Show the graph's node data frame; the
# `display` node attribute will show for
# each row, which node attribute value to
# display when the graph is rendered
graph %>%
  get_node_df()

# This function can be called multiple
# times on a graph; after the first time
# (i.e., creation of the `display`
# attribute), the `default` value won't
# be used
graph %>%
  set_node_attr_to_display(
    nodes = 4,
    attr = label) %>%
  set_node_attr_to_display(
    nodes = c(1, 3),
    attr = id) %>%
  get_node_df()
```

`set_node_attr_w_fcn` *Set node attribute values with a graph function*

Description

From a graph object of class dgr_graph or a node data frame, set node attribute properties for all nodes in the graph using one of several whole-graph functions.

Usage

```
set_node_attr_w_fcn(graph, node_attr_fcn, ..., column_name = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
node_attr_fcn	the name of the function to use for creating a column of node attribute values. Valid functions are: get_alpha_centrality, get_authority_centrality, get_betweenness, get_bridging, get_closeness, get_cmty_edge_btwns, get_cmty_fast_greedy, get_cmty_l_eigenvec, get_cmty_louvain, get_cmty_walktrap, get_constraint, get_degree_distribution, get_degree_histogram, get_degree_in, get_degree_out, get_degree_total, get_eccentricity, get_eigen_centrality, get_pagerank, get_s_connected_cmpts, and get_w_connected_cmpts.
...	arguments and values to pass to the named function in node_attr_fcn, if necessary.
column_name	an option to supply a column name for the new node attribute column. If NULL then the column name supplied by the function will used along with a __A suffix.

Value

either a graph object of class dgr_graph.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 10,
    m = 22,
    set_seed = 23) %>%
  set_node_attrs(
    node_attr = value,
    values = rnorm(
      n = count_nodes(.),
      mean = 5,
      sd = 1) %>% round(1))

# Get the betweenness values for
# each of the graph's nodes as a
# node attribute
graph_1 <-
  graph %>%
  set_node_attr_w_fcn(
```

```
    node_attr_fcn = "get_betweenness")

# Inspect the graph's internal
# node data frame
graph_1 %>%
  get_node_df()

# If a specified function takes argument
# values, these can be supplied as well
graph_2 <-
  graph %>%
  set_node_attr_w_fcn(
    node_attr_fcn = "get_alpha_centrality",
    alpha = 2,
    exo = 2)

# Inspect the graph's internal
# node data frame
graph_2 %>%
  get_node_df()

# The new column name can be provided
graph_3 <-
  graph %>%
  set_node_attr_w_fcn(
    node_attr_fcn = "get_pagerank",
    column_name = "pagerank")

# Inspect the graph's internal
# node data frame
graph_3 %>%
  get_node_df()

# If `graph_3` is modified by
# adding a new node then the column
# `pagerank` will have stale data; we
# can run the function again and re-use
# the existing column name to provide
# updated values
graph_3 <-
  graph_3 %>%
  add_node(
    from = 1,
    to = 3) %>%
  set_node_attr_w_fcn(
    node_attr_fcn = "get_pagerank",
    column_name = "pagerank")

# Inspect the graph's internal
# node data frame
graph_3 %>%
  get_node_df()
```

set_node_position *Apply a layout position to a single node*

Description

Apply position information for a single node. This is done by setting the x and y attrs for a node id or node label supplied in node. When rendering the graph, nodes with attribute values set for x and y will be fixed to those positions on the graph canvas.

Usage

```
set_node_position(graph, node, x, y, use_labels = FALSE)
```

Arguments

graph	a graph object of class dgr_graph.
node	a single-length vector containing either a node ID value (integer) or a node label (character) for which position information should be applied.
x	the x coordinate to set for the node.
y	the y coordinate to set for the node.
use_labels	an option to use a node label value in node. Note that this is only possible if all nodes have distinct label values set and none exist as an NA value.

Value

a graph object of class dgr_graph.

Examples

```
# Create a simple graph with 4 nodes
graph <-
  create_graph() %>%
  add_node(label = "one") %>%
  add_node(label = "two") %>%
  add_node(label = "three") %>%
  add_node(label = "four")

# Add position information to each of
# the graph's nodes
graph <-
  graph %>%
  set_node_position(
    node = 1,
    x = 1, y = 1) %>%
  set_node_position(
    node = 2,
    x = 2, y = 2) %>%
  set_node_position(
```

```
    node = 3,
    x = 3, y = 3) %>%
  set_node_position(
    node = 4,
    x = 4, y = 4)

# View the graph's node data frame to
# verify that the `x` and `y` node
# attributes are available and set to
# the values provided
graph %>%
  get_node_df()

# The same function can modify the data
# in the `x` and `y` attributes
graph <-
  graph %>%
  set_node_position(
    node = 1,
    x = 1, y = 4) %>%
  set_node_position(
    node = 2,
    x = 3, y = 3) %>%
  set_node_position(
    node = 3,
    x = 3, y = 2) %>%
  set_node_position(
    node = 4,
    x = 4, y = 1)

# View the graph's node data frame
graph %>%
  get_node_df()

# Position changes can also be made by
# supplying a node `label` value (and setting
# `use_labels` to TRUE). For this to work,
# all `label` values in the graph's ndf must
# be unique and non-NA
graph <-
  graph %>%
  set_node_position(
    node = "one",
    x = 1, y = 1,
    use_labels = TRUE) %>%
  set_node_position(
    node = "two",
    x = 2, y = 2,
    use_labels = TRUE)

# View the graph's node data frame
graph %>%
  get_node_df()
```

to_igraph *Convert a DiagrammeR graph to an igraph one*

Description

Convert a DiagrammeR graph to an igraph graph object.

Usage

```
to_igraph(graph)
```

Arguments

graph	a graph object of class dgr_graph.

Value

an igraph object.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 36,
    m = 50,
    set_seed = 23)

# Confirm that `graph` is a
# DiagrammeR graph by getting
# the object's class
class(graph)

# Convert the DiagrammeR graph
# to an igraph object
ig_graph <- to_igraph(graph)

# Get the class of the converted
# graph, just to be certain
class(ig_graph)

# Get a summary of the igraph
# graph object
summary(ig_graph)
```

transform_to_complement_graph
Create a complement of a graph

Description

Create a complement graph which contains only edges not present in the input graph. It's important to nodes that any edge attributes in the input graph's edges will be lost. Node attributes will be retained, since they are not affected by this transformation.

Usage

```
transform_to_complement_graph(graph, loops = FALSE)
```

Arguments

graph	a graph object of class dgr_graph that is created using create_graph.
loops	an option for whether loops should be generated in the complement graph.

Value

a graph object of class dgr_graph.

Examples

```
# Create a simple graph
# with a single cycle
graph <-
  create_graph() %>%
  add_cycle(n = 4)

# Get the graph's edge
# data frame
graph %>%
  get_edge_df()

# Create the complement
# of the graph
graph_c <-
  graph %>%
    transform_to_complement_graph()

# Get the edge data frame
# for the complement graph
graph_c %>%
  get_edge_df()
```

`transform_to_min_spanning_tree` *Get a minimum spanning tree subgraph*

Description

Get a minimum spanning tree subgraph for a connected graph of class `dgr_graph`.

Usage

```
transform_to_min_spanning_tree(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 10,
    m = 15,
    set_seed = 23)

# Obtain Jaccard similarity
# values for each pair of
# nodes as a square matrix
j_sim_matrix <-
  graph %>%
    get_jaccard_similarity()

# Create a weighted, undirected
# graph from the resultant matrix
# (effectively treating that
# matrix as an adjacency matrix)
graph <-
  j_sim_matrix %>%
  from_adj_matrix(weighted = TRUE)

# The graph in this case is a fully connected graph
# with loops, where jaccard similarity values are
# assigned as edge weights (edge attribute `weight`);
# The minimum spanning tree for this graph is the
```

```
# connected subgraph where the edges retained have
# the lowest similarity values possible
min_spanning_tree_graph <-
  graph %>%
  transform_to_min_spanning_tree() %>%
  copy_edge_attrs(
    edge_attr_from = weight,
    edge_attr_to = label) %>%
  set_edge_attrs(
    edge_attr = fontname,
    values = "Helvetica") %>%
  set_edge_attrs(
    edge_attr = color,
    values = "gray85") %>%
  rescale_edge_attrs(
    edge_attr_from = weight,
    to_lower_bound = 0.5,
    to_upper_bound = 4.0,
      edge_attr_to = penwidth)
```

`transform_to_subgraph_ws` *Create a subgraph using node/edge selection*

Description

Create a subgraph based on a selection of nodes or edges stored in the graph object. Selections of nodes can be performed using the following `select_...` functions: `select_nodes()`, `select_last_nodes_created()`, `select_nodes_by_degree()`, `select_nodes_by_id()`, or `select_nodes_in_neighbo`
Alternatively, selections of edges can be made with these functions: `select_edges()`, `select_last_edge()`, or `select_edges_by_node_id()`. Selections of nodes or edges can also be performed using any of the traversal functions (`trav_...`).

Usage

```
transform_to_subgraph_ws(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph` that is created using `create_graph`.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a node data frame (ndf)
ndf <-
  create_node_df(
    n = 6,
    value =
      c(3.5, 2.6, 9.4,
        2.7, 5.2, 2.1))

# Create an edge data frame (edf)
edf <-
  create_edge_df(
    from = c(1, 2, 4, 5, 2, 6),
      to = c(2, 4, 1, 3, 5, 5))

# Create a graph
graph <-
  create_graph(
    nodes_df = ndf,
    edges_df = edf)

# Create a selection of nodes, this selects
# nodes `1`, `3`, and `5`
graph <-
  graph %>%
  select_nodes(
    conditions = value > 3)

# Create a subgraph based on the selection
subgraph <-
  graph %>%
  transform_to_subgraph_ws()

# Display the graph's node data frame
subgraph %>%
  get_node_df()

# Display the graph's edge data frame
subgraph %>%
  get_edge_df()
```

---

trav_both *Traverse from one or more selected nodes onto neighboring nodes*

Description

From a graph object of class dgr_graph move from one or more nodes present in a selection to other nodes that are connected by edges, replacing the current nodes in the selection with those nodes traversed to. An optional filter by node attribute can limit the set of nodes traversed to.

Usage

```
trav_both(graph, conditions = NULL, copy_attrs_from = NULL,
  copy_attrs_as = NULL, agg = "sum", add_to_selection = FALSE)
```

Arguments

graph	a graph object of class dgr_graph.
conditions	an option to use filtering conditions for the traversal.
copy_attrs_from	providing a node attribute name will copy those node attribute values to the traversed nodes. Any values extant on the nodes traversed to will be replaced.
copy_attrs_as	if a node attribute name is provided in copy_attrs_from, this option will allow the copied attribute values to be written under a different attribute name. If the attribute name provided in copy_attrs_as does not exist in the graph's ndf, the new node attribute will be created with the chosen name.
agg	if a node attribute is provided to copy_attrs_from, then an aggregation function is required since there may be cases where multiple edge attribute values will be passed onto the traversed node(s). To pass only a single value, the following aggregation functions can be used: sum, min, max, mean, or median.
add_to_selection	an option to either add the traversed to nodes to the active selection of nodes (TRUE) or switch the active selection entirely to those traversed to nodes (FALSE, the default case).

Value

a graph object of class dgr_graph.

Examples

```
# Set a seed
set.seed(23)

# Create a simple graph
graph <-
  create_graph() %>%
  add_n_nodes(
    n = 2,
    type = "a",
    label = c("asd", "iekd")) %>%
  add_n_nodes(
    n = 3,
    type = "b",
    label = c("idj", "edl", "ohd")) %>%
  add_edges_w_string(
    edges = "1->2 1->3 2->4 2->5 3->5",
    rel = c(NA, "A", "B", "C", "D"))

# Create a data frame with node ID values
```

```
# representing the graph edges (with `from`
# and `to` columns), and, a set of numeric values
df_edges <-
  data.frame(
    from = c(1, 1, 2, 2, 3),
    to = c(2, 3, 4, 5, 5),
    values = round(rnorm(5, 5), 2))

# Create a data frame with node ID values
# representing the graph nodes (with the `id`
# columns), and, a set of numeric values
df_nodes <-
  data.frame(
    id = 1:5,
    values = round(rnorm(5, 7), 2))

# Join the data frame to the graph's internal
# edge data frame (edf)
graph <-
  graph %>%
  join_edge_attrs(df = df_edges) %>%
  join_node_attrs(df = df_nodes)

# Show the graph's internal node data frame
graph %>%
  get_node_df()

# Show the graph's internal edge data frame
graph %>%
  get_edge_df()

# Perform a simple traversal from node `3`
# to adjacent nodes with no conditions on
# the nodes traversed to
graph %>%
  select_nodes_by_id(nodes = 3) %>%
  trav_both() %>%
  get_selection()

# Traverse from node `2` to any adjacent
# nodes, filtering to those nodes that have
# numeric values less than `8.0` for
# the `values` node attribute
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_both(
    conditions = values < 8.0) %>%
  get_selection()

# Traverse from node `5` to any adjacent
# nodes, filtering to those nodes that
# have a `type` attribute of `b`
graph %>%
```

```
  select_nodes_by_id(nodes = 5) %>%
  trav_both(
    conditions = type == "b") %>%
  get_selection()

# Traverse from node `2` to any adjacent
# nodes, and use multiple conditions for the
# traversal
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_both(
    conditions =
      type == "a" &
      values > 8.0) %>%
  get_selection()

# Traverse from node `2` to any adjacent
# nodes, and use multiple conditions with
# a single-length vector
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_both(
    conditions =
      type == "a" | values > 8.0) %>%
  get_selection()

# Traverse from node `2` to any adjacent
# nodes, and use a regular expression as
# a filtering condition
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_both(
    conditions = grepl("..d", label)) %>%
  get_selection()

# Create another simple graph to demonstrate
# copying of node attribute values to traversed
# nodes
graph <-
  create_graph() %>%
  add_path(n = 5) %>%
  select_nodes_by_id(nodes = c(2, 4)) %>%
  set_node_attrs_ws(
    node_attr = value,
    value = 5)

# Show the graph's internal node data frame
graph %>%
  get_node_df()

# Show the graph's internal edge data frame
graph %>%
  get_edge_df()
```

```
# Perform a traversal from the inner nodes
# (`2` and `4`) to their adjacent nodes (`1`,
# `3`, and `5`) while also applying the node
# attribute `value` to target nodes; node `3`
# will obtain a `value` of 10 since a traversal
# to `3` will occur from `2` and `4` (and
# multiple values passed will be summed)
graph <-
  graph %>%
  trav_both(
    copy_attrs_from = value,
    agg = "sum")

# Show the graph's internal node data frame
# after this change
graph %>%
  get_node_df()
```

`trav_both_edge` *Traverse from one or more selected nodes onto adjacent edges*

Description

From a graph object of class `dgr_graph` move to adjacent edges from a selection of one or more selected nodes, thereby creating a selection of edges. An optional filter by edge attribute can limit the set of edges traversed to.

Usage

```
trav_both_edge(graph, conditions = NULL, copy_attrs_from = NULL,
  copy_attrs_as = NULL, agg = "sum")
```

Arguments

`graph`	a graph object of class `dgr_graph`.
`conditions`	an option to use filtering conditions for the traversal.
`copy_attrs_from`	providing a node attribute name will copy those node attribute values to the traversed edges. If the edge attribute already exists, the values will be merged to the traversed edges; otherwise, a new edge attribute will be created.
`copy_attrs_as`	if a node attribute name is provided in `copy_attrs_from`, this option will allow the copied attribute values to be written under a different edge attribute name. If the attribute name provided in `copy_attrs_as` does not exist in the graph's edf, the new edge attribute will be created with the chosen name.
`agg`	if a node attribute is provided to `copy_attrs_from`, then an aggregation function is required since there may be cases where multiple node attribute values will be passed onto the traversed edge(s). To pass only a single value, the following aggregation functions can be used: `sum`, `min`, `max`, `mean`, or `median`.

Value

a graph object of class dgr_graph.

Examples

```
# Set a seed
set.seed(23)

# Create a simple graph
graph <-
  create_graph() %>%
  add_n_nodes(
    n = 2,
    type = "a",
    label = c("asd", "iekd")) %>%
  add_n_nodes(
    n = 3,
    type = "b",
    label = c("idj", "edl", "ohd")) %>%
  add_edges_w_string(
    edges = "1->2 1->3 2->4 2->5 3->5",
    rel = c(NA, "A", "B", "C", "D"))

# Create a data frame with node ID values
# representing the graph edges (with `from`
# and `to` columns), and, a set of numeric values
df <-
  data.frame(
    from = c(1, 1, 2, 2, 3),
    to = c(2, 3, 4, 5, 5),
    values = round(rnorm(5, 5), 2))

# Join the data frame to the graph's internal
# edge data frame (edf)
graph <-
  graph %>%
  join_edge_attrs(df = df)

# Show the graph's internal edge data frame
graph %>%
  get_edge_df()

# Perform a simple traversal from nodes to
# adjacent edges with no conditions on the
# nodes traversed to
graph %>%
  select_nodes_by_id(nodes = 3) %>%
  trav_both_edge() %>%
  get_selection()

# Traverse from node `2` to any adjacent
# edges, filtering to those edges that have
```

```
# NA values for the `rel` edge attribute
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_both_edge(
    conditions = is.na(rel)) %>%
  get_selection()

# Traverse from node `2` to any adjacent
# edges, filtering to those edges that have
# numeric values greater than `6.5` for
# the `rel` edge attribute
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_both_edge(
    conditions = values > 6.5) %>%
  get_selection()

# Traverse from node `5` to any adjacent
# edges, filtering to those edges that
# have values equal to `C` for the `rel`
# edge attribute
graph %>%
  select_nodes_by_id(nodes = 5) %>%
  trav_both_edge(
    conditions = rel == "C") %>%
  get_selection()

# Traverse from node `2` to any adjacent
# edges, filtering to those edges that
# have values in the set `B` and `C` for
# the `rel` edge attribute
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_both_edge(
    conditions = rel %in% c("B", "C")) %>%
  get_selection()

# Traverse from node `2` to any adjacent
# edges, and use multiple conditions for the
# traversal
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_both_edge(
    conditions =
      rel %in% c("B", "C") &
      values > 4.0) %>%
  get_selection()

# Traverse from node `2` to any adjacent
# edges, and use multiple conditions with
# a single-length vector
graph %>%
  select_nodes_by_id(nodes = 2) %>%
```

```
  trav_both_edge(
    conditions =
      rel %in% c("B", "C") |
      values > 4.0) %>%
  get_selection()

# Traverse from node `2` to any adjacent
# edges, and use a regular expression as
# a filtering condition
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_both_edge(
    conditions = grepl("B|C", rel)) %>%
  get_selection()

# Create another simple graph to demonstrate
# copying of node attribute values to traversed
# edges
graph <-
  create_graph() %>%
  add_path(n = 4) %>%
  select_nodes_by_id(nodes = 2:3) %>%
  set_node_attrs_ws(
    node_attr = value,
    value = 5)

# Show the graph's internal edge data frame
graph %>%
  get_edge_df()

# Show the graph's internal node data frame
graph %>%
  get_node_df()

# Perform a traversal from the nodes to
# the adjacent edges while also applying
# the node attribute `value` to the edges (in
# this case summing the `value` of 5 from
# all contributing nodes adding as an edge
# attribute)
graph <-
  graph %>%
  trav_both_edge(
    copy_attrs_from = value,
    agg = "sum")

# Show the graph's internal edge data frame
# after this change
graph %>%
  get_edge_df()
```

trav_in *Traverse from one or more selected nodes onto adjacent, inward nodes*

Description

From a graph object of class dgr_graph move along inward edges from one or more nodes present in a selection to other connected nodes, replacing the current nodes in the selection with those nodes traversed to. An optional filter by node attribute can limit the set of nodes traversed to.

Usage

```
trav_in(graph, conditions = NULL, copy_attrs_from = NULL,
  copy_attrs_as = NULL, agg = "sum", add_to_selection = FALSE)
```

Arguments

graph	a graph object of class dgr_graph.
conditions	an option to use filtering conditions for the traversal.
copy_attrs_from	providing a node attribute name will copy those node attribute values to the traversed nodes. Any values extant on the nodes traversed to will be replaced.
copy_attrs_as	if a node attribute name is provided in copy_attrs_from, this option will allow the copied attribute values to be written under a different attribute name. If the attribute name provided in copy_attrs_as does not exist in the graph's ndf, the new node attribute will be created with the chosen name.
agg	if a node attribute is provided to copy_attrs_from, then an aggregation function is required since there may be cases where multiple edge attribute values will be passed onto the traversed node(s). To pass only a single value, the following aggregation functions can be used: sum, min, max, mean, or median.
add_to_selection	an option to either add the traversed to nodes to the active selection of nodes (TRUE) or switch the active selection entirely to those traversed to nodes (FALSE, the default case).

Value

a graph object of class dgr_graph.

Examples

```
# Set a seed
set.seed(23)

# Create a simple graph
graph <-
  create_graph() %>%
```

```
  add_n_nodes(
    n = 2,
    type = "a",
    label = c("asd", "iekd")) %>%
  add_n_nodes(
    n = 3,
    type = "b",
    label = c("idj", "edl", "ohd")) %>%
  add_edges_w_string(
    edges = "1->2 1->3 2->4 2->5 3->5",
    rel = c(NA, "A", "B", "C", "D"))

# Create a data frame with node ID values
# representing the graph edges (with `from`
# and `to` columns), and, a set of numeric values
df_edges <-
  data.frame(
    from = c(1, 1, 2, 2, 3),
    to = c(2, 3, 4, 5, 5),
    values = round(rnorm(5, 5), 2))

# Create a data frame with node ID values
# representing the graph nodes (with the `id`
# columns), and, a set of numeric values
df_nodes <-
  data.frame(
    id = 1:5,
    values = round(rnorm(5, 7), 2))

# Join the data frame to the graph's internal
# edge data frame (edf)
graph <-
  graph %>%
  join_edge_attrs(df = df_edges) %>%
  join_node_attrs(df = df_nodes)

# Show the graph's internal node data frame
graph %>%
  get_node_df()

# Show the graph's internal edge data frame
graph %>%
  get_edge_df()

# Perform a simple traversal from node `4` to
# inward adjacent edges with no conditions
# on the nodes traversed to
graph %>%
  select_nodes_by_id(nodes = 4) %>%
  trav_in() %>%
  get_selection()

# Traverse from node `5` to inbound-facing
```

```
# nodes, filtering to those nodes that have
# numeric values greater than `5.0` for
# the `values` node attribute
graph %>%
  select_nodes_by_id(nodes = 4) %>%
  trav_in(
    conditions = values > 5.0) %>%
  get_selection()

# Traverse from node `5` to any inbound
# nodes, filtering to those nodes that
# have a `type` attribute of `b`
graph %>%
  select_nodes_by_id(nodes = 5) %>%
  trav_in(
    conditions = type == "b") %>%
  get_selection()

# Traverse from node `5` to any inbound
# nodes, filtering to those nodes that
# have a degree of `2`
graph %>%
  {
  node_degrees <-
    get_node_info(.) %>%
    dplyr::select(id, deg)
  join_node_attrs(., node_degrees)
  } %>%
  select_nodes_by_id(nodes = 5) %>%
  trav_in(
    conditions = deg == 2) %>%
  get_selection()

# Traverse from node `5` to any inbound
# nodes, and use multiple conditions for the
# traversal
graph %>%
  select_nodes_by_id(nodes = 5) %>%
  trav_in(
    conditions =
      type == "a" &
      values > 6.0) %>%
  get_selection()

# Traverse from node `5` to any inbound
# nodes, and use multiple conditions with
# a single-length vector
graph %>%
  select_nodes_by_id(nodes = 5) %>%
  trav_in(
    conditions =
      type == "b" | values > 6.0) %>%
  get_selection()
```

```
# Traverse from node `5` to any inbound
# nodes, and use a regular expression as
# a filtering condition
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_in(
    conditions = grepl("^i.*", label)) %>%
  get_selection()

# Create another simple graph to demonstrate
# copying of node attribute values to traversed
# nodes
graph <-
  create_graph() %>%
  add_node() %>%
  select_nodes() %>%
  add_n_nodes_ws(
    n = 2,
    direction = "from") %>%
  clear_selection() %>%
  select_nodes_by_id(nodes = 2:3) %>%
  set_node_attrs_ws(
    node_attr = value,
    value = 5)

# Show the graph's internal node data frame
graph %>%
  get_node_df()

# Show the graph's internal edge data frame
graph %>%
  get_edge_df()

# Perform a traversal from the outer nodes
# (`2` and `3`) to the central node (`1`) while
# also applying the node attribute `value` to
# node `1` (summing the `value` of 5 from
# both nodes before applying the value to the
# target node)
graph <-
  graph %>%
  trav_in(
    copy_attrs_from = value,
    agg = "sum")

# Show the graph's internal node data frame
# after this change
graph %>%
  get_node_df()
```

trav_in_edge *Traverse from one or more selected nodes onto adjacent, inward edges*

Description

From a graph object of class dgr_graph move to incoming edges from a selection of one or more selected nodes, thereby creating a selection of edges. An optional filter by edge attribute can limit the set of edges traversed to.

Usage

```
trav_in_edge(graph, conditions = NULL, copy_attrs_from = NULL,
  copy_attrs_as = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
conditions	an option to use filtering conditions for the traversal.
copy_attrs_from	providing a node attribute name will copy those node attribute values to the traversed edges. If the edge attribute already exists, the values will be merged to the traversed edges; otherwise, a new edge attribute will be created.
copy_attrs_as	if a node attribute name is provided in copy_attrs_from, this option will allow the copied attribute values to be written under a different edge attribute name. If the attribute name provided in copy_attrs_as does not exist in the graph's edf, the new edge attribute will be created with the chosen name.

Value

a graph object of class dgr_graph.

Examples

```
# Set a seed
set.seed(23)

# Create a simple graph
graph <-
  create_graph() %>%
  add_n_nodes(
    n = 2,
    type = "a",
    label = c("asd", "iekd")) %>%
  add_n_nodes(
    n = 3,
    type = "b",
    label = c("idj", "edl", "ohd")) %>%
  add_edges_w_string(
```

```
    edges = "1->2 1->3 2->4 2->5 3->5",
    rel = c(NA, "A", "B", "C", "D"))

# Create a data frame with node ID
# values representing the graph edges
# (with `from` and `to` columns), and,
# a set of numeric values
df <-
  data.frame(
    from = c(1, 1, 2, 2, 3),
    to = c(2, 3, 4, 5, 5),
    values = round(rnorm(5, 5), 2))

# Join the data frame to the graph's
# internal edge data frame (edf)
graph <-
  graph %>%
  join_edge_attrs(df = df)

# Show the graph's internal edge data frame
graph %>%
  get_edge_df()

# Perform a simple traversal from
# nodes to inbound edges with no
# conditions on the nodes
# traversed to
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_in_edge() %>%
  get_selection()

# Traverse from node `2` to any
# inbound edges, filtering to
# those edges that have NA values
# for the `rel` edge attribute
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_in_edge(
    conditions = is.na(rel)) %>%
  get_selection()

# Traverse from node `2` to any
# inbound edges, filtering to those
# edges that do not have NA values
# for the `rel` edge attribute
# (since there are no allowed
# traversals, the selection of node
# `2` is retained)
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_in_edge(
    conditions = !is.na(rel)) %>%
```

```
  get_selection()

# Traverse from node `5` to any
# inbound edges, filtering to those
# edges that have numeric values
# greater than `5.5` for the `rel`
# edge attribute
graph %>%
  select_nodes_by_id(nodes = 5) %>%
  trav_in_edge(
    conditions = values > 5.5) %>%
  get_selection()

# Traverse from node `5` to any
# inbound edges, filtering to those
# edges that have values equal to
# `D` for the `rel` edge attribute
graph %>%
  select_nodes_by_id(nodes = 5) %>%
  trav_in_edge(
    conditions = rel == "D") %>%
  get_selection()

# Traverse from node `5` to any
# inbound edges, filtering to those
# edges that have values in the set
# `C` and `D` for the `rel` edge
# attribute
graph %>%
  select_nodes_by_id(nodes = 5) %>%
  trav_in_edge(
    conditions = rel %in% c("C", "D")) %>%
  get_selection()

# Traverse from node `5` to any
# inbound edges, and use multiple
# conditions for the traversal
graph %>%
  select_nodes_by_id(nodes = 5) %>%
  trav_in_edge(
    conditions =
      rel %in% c("C", "D") &
      values > 5.5) %>%
  get_selection()

# Traverse from node `5` to any
# inbound edges, and use multiple
# conditions with a single-length
# vector
graph %>%
  select_nodes_by_id(nodes = 5) %>%
  trav_in_edge(
    conditions =
```

```
      rel %in% c("D", "E") |
      values > 5.5) %>%
  get_selection()

# Traverse from node `5` to any
# inbound edges, and use a regular
# expression as a filtering condition
graph %>%
  select_nodes_by_id(nodes = 5) %>%
  trav_in_edge(
    conditions = grepl("C|D", rel)) %>%
  get_selection()

# Show the graph's internal ndf
graph %>%
  get_node_df()

# Show the graph's internal edf
graph %>%
  get_edge_df()

# Perform a traversal from all
# nodes to their incoming edges and,
# while doing so, copy the `label`
# node attribute to any of the nodes'
# incoming edges
graph <-
  graph %>%
  select_nodes() %>%
  trav_in_edge(
    copy_attrs_from = label)

# Show the graph's internal edge
# data frame after this change
graph %>%
  get_edge_df()
```

`trav_in_node` *Traverse from one or more selected edges onto adjacent, inward nodes*

Description

From a graph object of class `dgr_graph` with an active selection of edges move with the edge direction to connected nodes, replacing the current edges in the selection with those nodes traversed to. An optional filter by node attribute can limit the set of nodes traversed to.

Usage

```
trav_in_node(graph, conditions = NULL, copy_attrs_from = NULL,
  copy_attrs_as = NULL, agg = "sum")
```

Arguments

graph	a graph object of class dgr_graph.
conditions	an option to use filtering conditions for the traversal.
copy_attrs_from	providing an edge attribute name will copy those edge attribute values to the traversed nodes. If the edge attribute already exists, the values will be merged to the traversed nodes; otherwise, a new node attribute will be created.
copy_attrs_as	if an edge attribute name is provided in copy_attrs_from, this option will allow the copied attribute values to be written under a different node attribute name. If the attribute name provided in copy_attrs_as does not exist in the graph's ndf, the new node attribute will be created with the chosen name.
agg	if an edge attribute is provided to copy_attrs_from, then an aggregation function is required since there may be cases where multiple edge attribute values will be passed onto the traversed node(s). To pass only a single value, the following aggregation functions can be used: sum, min, max, mean, or median.

Value

a graph object of class dgr_graph.

Examples

```
# Set a seed
set.seed(23)

# Create a simple graph
graph <-
  create_graph() %>%
  add_n_nodes(
    n = 2,
    type = "a",
    label = c("asd", "iekd")) %>%
  add_n_nodes(
    n = 3,
    type = "b",
    label = c("idj", "edl", "ohd")) %>%
  add_edges_w_string(
    edges = "1->2 1->3 2->4 2->5 3->5",
    rel = c(NA, "A", "B", "C", "D"))

# Create a data frame with node ID values
# representing the graph edges (with `from`
# and `to` columns), and, a set of numeric values
df_edges <-
  data.frame(
    from = c(1, 1, 2, 2, 3),
    to = c(2, 3, 4, 5, 5),
    values = round(rnorm(5, 5), 2))

# Create a data frame with node ID values
```

```
# representing the graph nodes (with the `id`
# columns), and, a set of numeric values
df_nodes <-
  data.frame(
    id = 1:5,
    values = round(rnorm(5, 7), 2))

# Join the data frame to the graph's internal
# edge data frame (edf)
graph <-
  graph %>%
  join_edge_attrs(df = df_edges) %>%
  join_node_attrs(df = df_nodes)

# Show the graph's internal node data frame
graph %>%
  get_node_df()

# Show the graph's internal edge data frame
graph %>%
  get_edge_df()

# Perform a simple traversal from the
# edge `1`->`3` to the attached node
# in the direction of the edge; here, no
# conditions are placed on the nodes
# traversed to
graph %>%
  select_edges(
    from = 1,
    to = 3) %>%
  trav_in_node() %>%
  get_selection()

# Traverse from edges `2`->`5` and
# `3`->`5` to the attached node along
# the direction of the edge; both
# traversals lead to the same node
graph %>%
  select_edges(
    from = 2,
    to = 5) %>%
  select_edges(
    from = 3,
    to = 5) %>%
  trav_in_node() %>%
  get_selection()

# Traverse from the edge `1`->`3`
# to the attached node where the edge
# is incoming, this time filtering
# numeric values greater than `5.0` for
# the `values` node attribute
```

```
graph %>%
  select_edges(
    from = 1,
    to = 3) %>%
  trav_in_node(
    conditions = values > 5.0) %>%
  get_selection()

# Traverse from the edge `1`->`3`
# to the attached node where the edge
# is incoming, this time filtering
# numeric values less than `5.0` for
# the `values` node attribute (the
# condition is not met so the original
# selection of edge `1` -> `3` remains)
graph %>%
  select_edges(
    from = 1,
    to = 3) %>%
  trav_in_node(
    conditions = values < 5.0) %>%
  get_selection()

# Traverse from the edge `1`->`2` to
# the node `2` using multiple conditions
# with a single-length vector
graph %>%
  select_edges(
    from = 1,
    to = 2) %>%
  trav_in_node(
    conditions =
      grepl(".*d$", label) |
      values < 6.0) %>%
  get_selection()

# Create another simple graph to demonstrate
# copying of edge attribute values to traversed
# nodes
graph <-
  create_graph() %>%
  add_node() %>%
  select_nodes() %>%
  add_n_nodes_ws(
    n = 2,
    direction = "to") %>%
  clear_selection() %>%
  select_nodes_by_id(nodes = 2) %>%
  set_node_attrs_ws(
    node_attr = value,
    value = 8) %>%
  clear_selection() %>%
  select_edges_by_edge_id(edges = 1) %>%
```

```
  set_edge_attrs_ws(
    edge_attr = value,
    value = 5) %>%
  clear_selection() %>%
  select_edges_by_edge_id(edges = 2) %>%
  set_edge_attrs_ws(
    edge_attr = value,
    value = 5) %>%
  clear_selection() %>%
  select_edges()

# Show the graph's internal edge data frame
graph %>%
  get_edge_df()

# Show the graph's internal node data frame
graph %>%
  get_node_df()

# Perform a traversal from the edges to
# the central node (`1`) while also applying
# the edge attribute `value` to the node (in
# this case summing the `value` of 5 from
# both edges before adding as a node attribute)
graph <-
  graph %>%
  trav_in_node(
    copy_attrs_from = value,
    agg = "sum")

# Show the graph's internal node data frame
# after this change
graph %>%
  get_node_df()
```

`trav_in_until` *Traverse inward node-by_node until stopping conditions are met*

Description

From a graph object of class `dgr_graph`, move along inward edges from one or more nodes present in a selection to other connected nodes, replacing the current nodes in the selection with those nodes traversed to until reaching nodes that satisfy one or more conditions.

Usage

```
trav_in_until(graph, conditions, max_steps = 30, exclude_unmatched = TRUE,
  add_to_selection = FALSE)
```

Arguments

graph	a graph object of class dgr_graph.
conditions	an option to use a stopping condition for the traversal. If the condition is met during the traversal (i.e., the node(s) traversed to match the condition), then those traversals will terminate at those nodes. Otherwise, traversals with continue and terminate when the number of steps provided in max_steps is reached.
max_steps	the maximum number of trav_in() steps (i.e., node-to-node traversals in the inward direction) to allow before stopping.
exclude_unmatched	if TRUE (the default value) then any nodes not satisfying the conditions provided in conditions that are in the ending selection are excluded.
add_to_selection	if TRUE then every node traversed will be part of the final selection of nodes. If FALSE (the default value) then only the nodes finally traversed to will be part of the final node selection.

Value

a graph object of class dgr_graph.

Examples

```
# Create a path graph and add
# values of 1 to 10 across the
# nodes from beginning to end;
# select the last path node
graph <-
  create_graph() %>%
  add_path(
    n = 10,
    node_data = node_data(
      value = 1:10)) %>%
  select_nodes_by_id(
    nodes = 10)

# Traverse inward, node-by-node
# until stopping at a node where
# the `value` attribute is 1
graph <-
  graph %>%
  trav_in_until(
    conditions =
      value == 1)

# Get the graph's node selection
graph %>%
  get_selection()

# Create two cycles in a graph and
# add values of 1 to 6 to the
```

```
# first cycle, and values 7 to
# 12 in the second; select nodes
# `6` and `12`
graph <-
  create_graph() %>%
  add_cycle(
    n = 6,
    node_data = node_data(
      value = 1:6)) %>%
  add_cycle(
    n = 6,
    node_data = node_data(
      value = 7:12)) %>%
  select_nodes_by_id(
    nodes = c(6, 12))

# Traverse inward, node-by-node
# from `6` and `12` until stopping
# at the first nodes where the
# `value` attribute is 1, 2, or 10;
# specify that we should only
# keep the finally traversed to
# nodes that satisfy the conditions
graph <-
  graph %>%
  trav_in_until(
    conditions =
      value %in% c(1, 2, 10),
    exclude_unmatched = TRUE)

# Get the graph's node selection
graph %>%
  get_selection()
```

`trav_out` *Traverse from one or more selected nodes onto adjacent, outward nodes*

Description

From a graph object of class `dgr_graph` move along outward edges from one or more nodes present in a selection to other connected nodes, replacing the current nodes in the selection with those nodes traversed to. An optional filter by node attribute can limit the set of nodes traversed to.

Usage

```
trav_out(graph, conditions = NULL, copy_attrs_from = NULL,
  copy_attrs_as = NULL, agg = "sum", add_to_selection = FALSE)
```

Arguments

graph	a graph object of class dgr_graph.
conditions	an option to use filtering conditions for the traversal.
copy_attrs_from	providing a node attribute name will copy those node attribute values to the traversed nodes. Any values extant on the nodes traversed to will be replaced.
copy_attrs_as	if a node attribute name is provided in copy_attrs_from, this option will allow the copied attribute values to be written under a different attribute name. If the attribute name provided in copy_attrs_as does not exist in the graph's ndf, the new node attribute will be created with the chosen name.
agg	if a node attribute is provided to copy_attrs_from, then an aggregation function is required since there may be cases where multiple edge attribute values will be passed onto the traversed node(s). To pass only a single value, the following aggregation functions can be used: sum, min, max, mean, or median.
add_to_selection	an option to either add the traversed to nodes to the active selection of nodes (TRUE) or switch the active selection entirely to those traversed to nodes (FALSE, the default case).

Value

a graph object of class dgr_graph.

Examples

```
# Set a seed
set.seed(23)

# Create a simple graph
graph <-
  create_graph() %>%
  add_n_nodes(
    n = 2,
    type = "a",
    label = c("asd", "iekd")) %>%
  add_n_nodes(
    n = 3,
    type = "b",
    label = c("idj", "edl", "ohd")) %>%
  add_edges_w_string(
    edges = "1->2 1->3 2->4 2->5 3->5",
    rel = c(NA, "A", "B", "C", "D"))

# Create a data frame with node ID values
# representing the graph edges (with `from`
# and `to` columns), and, a set of numeric values
df_edges <-
  data.frame(
    from = c(1, 1, 2, 2, 3),
```

```
    to = c(2, 3, 4, 5, 5),
    values = round(rnorm(5, 5), 2))

# Create a data frame with node ID values
# representing the graph nodes (with the `id`
# columns), and, a set of numeric values
df_nodes <-
  data.frame(
    id = 1:5,
    values = round(rnorm(5, 7), 2))

# Join the data frame to the graph's internal
# edge data frame (edf)
graph <-
  graph %>%
  join_edge_attrs(df = df_edges) %>%
  join_node_attrs(df = df_nodes)

# Show the graph's internal node data frame
graph %>%
  get_node_df()

# Show the graph's internal edge data frame
graph %>%
  get_edge_df()

# Perform a simple traversal from node `3`
# to outward adjacent nodes with no conditions
# on the nodes traversed to
graph %>%
  select_nodes_by_id(nodes = 3) %>%
  trav_out() %>%
  get_selection()

# Traverse from node `1` to outbound
# nodes, filtering to those nodes that have
# numeric values greater than `7.0` for
# the `values` node attribute
graph %>%
  select_nodes_by_id(nodes = 1) %>%
  trav_out(
    conditions = values > 7.0) %>%
  get_selection()

# Traverse from node `1` to any outbound
# nodes, filtering to those nodes that
# have a `type` attribute of `b`
graph %>%
  select_nodes_by_id(nodes = 1) %>%
  trav_out(
    conditions = type == "b") %>%
  get_selection()
```

```
# Traverse from node `2` to any outbound
# nodes, filtering to those nodes that
# have a degree of `1`
graph %>%
  {
  node_degrees <-
    get_node_info(.) %>%
    dplyr::select(id, deg)
  join_node_attrs(
    graph = .,
    df = node_degrees)
  } %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_out(
    conditions = deg == 1) %>%
  get_selection()

# Traverse from node `2` to any outbound
# nodes, and use multiple conditions for
# the traversal
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_out(
    conditions =
      type == "a" &
      values > 8.0) %>%
  get_selection()

# Traverse from node `2` to any
# outbound nodes, and use multiple
# conditions with a single-length vector
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_out(
    conditions =
      type == "b" |
      values > 8.0) %>%
  get_selection()

# Traverse from node `2` to any outbound
# nodes, and use a regular expression as
# a filtering condition
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_out(
    conditions = grepl("..d", label)) %>%
  get_selection()

# Create another simple graph to demonstrate
# copying of node attribute values to traversed
# nodes
graph <-
  create_graph() %>%
```

```
  add_node() %>%
  select_nodes() %>%
  add_n_nodes_ws(
    n = 2,
    direction = "to") %>%
  clear_selection() %>%
  select_nodes_by_id(nodes = 2:3) %>%
  set_node_attrs_ws(
    node_attr = value,
    value = 5)

# Show the graph's internal node data frame
graph %>%
  get_node_df()

# Show the graph's internal edge data frame
graph %>%
  get_edge_df()

# Perform a traversal from the outer nodes
# (`2` and `3`) to the central node (`1`) while
# also applying the node attribute `value` to
# node `1` (summing the `value` of 5 from
# both nodes before applying that value to the
# target node)
graph <-
  graph %>%
  trav_out(
    copy_attrs_from = value,
    agg = "sum")

# Show the graph's internal node data
# frame after this change
graph %>%
  get_node_df()
```

`trav_out_edge` *Traverse from one or more selected nodes onto adjacent, outward edges*

Description

From a graph object of class `dgr_graph` move to outgoing edges from a selection of one or more selected nodes, thereby creating a selection of edges. An optional filter by edge attribute can limit the set of edges traversed to.

Usage

```
trav_out_edge(graph, conditions = NULL, copy_attrs_from = NULL,
  copy_attrs_as = NULL)
```

Arguments

graph	a graph object of class dgr_graph.
conditions	an option to use filtering conditions for the traversal.
copy_attrs_from	providing a node attribute name will copy those node attribute values to the traversed edges. If the edge attribute already exists, the values will be merged to the traversed edges; otherwise, a new edge attribute will be created.
copy_attrs_as	if a node attribute name is provided in copy_attrs_from, this option will allow the copied attribute values to be written under a different edge attribute name. If the attribute name provided in copy_attrs_as does not exist in the graph's edf, the new edge attribute will be created with the chosen name.

Value

a graph object of class dgr_graph.

Examples

```
# Set a seed
set.seed(23)

# Create a simple graph
graph <-
  create_graph() %>%
  add_n_nodes(
    n = 2,
    type = "a",
    label = c("asd", "iekd")) %>%
  add_n_nodes(
    n = 3,
    type = "b",
    label = c("idj", "edl", "ohd")) %>%
  add_edges_w_string(
    edges = "1->2 1->3 2->4 2->5 3->5",
    rel = c(NA, "A", "B", "C", "D")) %>%
  set_node_attrs(
    node_attr = values,
    values = c(2.3, 4.7, 9.4,
               8.3, 6.3))

# Create a data frame with node ID values
# representing the graph edges (with `from`
# and `to` columns), and, a set of numeric values
df <-
  data.frame(
    from = c(1, 1, 2, 2, 3),
    to = c(2, 3, 4, 5, 5),
    values = round(rnorm(5, 5), 2))

# Join the data frame to the graph's internal
```

```
# edge data frame (edf)
graph <-
  graph %>%
  join_edge_attrs(
    df = df)

# Show the graph's internal node data frame
graph %>%
  get_node_df()

# Show the graph's internal edge data frame
graph %>%
  get_edge_df()

# Perform a simple traversal from nodes to
# outbound edges with no conditions on the
# nodes traversed to
graph %>%
  select_nodes_by_id(nodes = 1) %>%
  trav_out_edge() %>%
  get_selection()

# Traverse from node `1` to any outbound
# edges, filtering to those edges that have
# NA values for the `rel` edge attribute
graph %>%
  select_nodes_by_id(nodes = 1) %>%
  trav_out_edge(
    conditions = is.na(rel)) %>%
  get_selection()

# Traverse from node `3` to any outbound
# edges, filtering to those edges that have
# numeric values greater than `5.0` for
# the `rel` edge attribute
graph %>%
  select_nodes_by_id(nodes = 3) %>%
  trav_out_edge(
    conditions = values > 5.0) %>%
  get_selection()

# Traverse from node `1` to any outbound
# edges, filtering to those edges that
# have values equal to `A` for the `rel`
# edge attribute
graph %>%
  select_nodes_by_id(nodes = 1) %>%
  trav_out_edge(
    conditions = rel == "A") %>%
  get_selection()

# Traverse from node `2` to any outbound
# edges, filtering to those edges that
```

```
# have values in the set `B` and `C` for
# the `rel` edge attribute
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_out_edge(
    conditions = rel %in% c("B", "C")) %>%
  get_selection()

# Traverse from node `2` to any
# outbound edges, and use multiple
# conditions for the traversal
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_out_edge(
    conditions =
      rel %in% c("B", "C") &
      values >= 5.0) %>%
  get_selection()

# Traverse from node `2` to any
# outbound edges, and use multiple
# conditions
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_out_edge(
    conditions =
      rel %in% c("B", "C") |
      values > 6.0) %>%
  get_selection()

# Traverse from node `2` to any outbound
# edges, and use a regular expression as
# a filtering condition
graph %>%
  select_nodes_by_id(nodes = 2) %>%
  trav_out_edge(
    conditions = grepl("B|C", rel)) %>%
  get_selection()

# Perform a traversal from all nodes to
# their outgoing edges and, while doing
# so, copy the `label` node attribute
# to any of the nodes' incoming edges
graph <-
  graph %>%
  select_nodes() %>%
  trav_out_edge(
    copy_attrs_from = label)

# Show the graph's internal edge
# data frame after this change
graph %>%
  get_edge_df()
```

trav_out_node *Traverse from one or more selected edges onto adjacent, outward nodes*

Description

From a graph object of class dgr_graph with an active selection of edges move opposite to the edge direction to connected nodes, replacing the current edge selection with those nodes traversed to. An optional filter by node attribute can limit the set of nodes traversed to.

Usage

```
trav_out_node(graph, conditions = NULL, copy_attrs_from = NULL,
  copy_attrs_as = NULL, agg = "sum")
```

Arguments

graph	a graph object of class dgr_graph.
conditions	an option to use filtering conditions for the traversal.
copy_attrs_from	providing an edge attribute name will copy those edge attribute values to the traversed nodes. If the edge attribute already exists, the values will be merged to the traversed nodes; otherwise, a new node attribute will be created.
copy_attrs_as	if an edge attribute name is provided in copy_attrs_from, this option will allow the copied attribute values to be written under a different node attribute name. If the attribute name provided in copy_attrs_as does not exist in the graph's ndf, the new node attribute will be created with the chosen name.
agg	if an edge attribute is provided to copy_attrs_from, then an aggregation function is required since there may be cases where multiple edge attribute values will be passed onto the traversed node(s). To pass only a single value, the following aggregation functions can be used: sum, min, max, mean, or median.

Value

a graph object of class dgr_graph.

Examples

```
# Set a seed
set.seed(23)

# Create a simple graph
graph <-
  create_graph() %>%
  add_n_nodes(
    n = 2,
    type = "a",
```

```
    label = c("asd", "iekd")) %>%
  add_n_nodes(
    n = 3,
    type = "b",
    label = c("idj", "edl", "ohd")) %>%
  add_edges_w_string(
    edges = "1->2 1->3 2->4 2->5 3->5",
    rel = c(NA, "A", "B", "C", "D"))

# Create a data frame with node ID values
# representing the graph edges (with `from`
# and `to` columns), and, a set of numeric values
df_edges <-
  data.frame(
    from = c(1, 1, 2, 2, 3),
      to = c(2, 3, 4, 5, 5),
    values = round(rnorm(5, 5), 2))

# Create a data frame with node ID values
# representing the graph nodes (with the `id`
# columns), and, a set of numeric values
df_nodes <-
  data.frame(
    id = 1:5,
    values = round(rnorm(5, 7), 2))

# Join the data frame to the graph's internal
# edge data frame (edf)
graph <-
  graph %>%
  join_edge_attrs(df = df_edges) %>%
  join_node_attrs(df = df_nodes)

# Show the graph's internal node data frame
graph %>%
  get_node_df()

# Show the graph's internal edge data frame
graph %>%
  get_edge_df()

# Perform a simple traversal from the
# edge `1`->`3` to the attached node
# in the direction of the edge; here, no
# conditions are placed on the nodes
# traversed to
graph %>%
  select_edges(
    from = 1,
      to = 3) %>%
  trav_out_node() %>%
  get_selection()
```

```
# Traverse from edges `2`->`5` and
# `3`->`5` to the attached node along
# the direction of the edge; here, the
# traversals lead to different nodes
graph %>%
  select_edges(
    from = 2,
      to = 5) %>%
  select_edges(
    from = 3,
      to = 5) %>%
  trav_out_node() %>%
  get_selection()

# Traverse from the edge `1`->`3`
# to the attached node where the edge
# is outgoing, this time filtering
# numeric values greater than `7.0` for
# the `values` node attribute
graph %>%
  select_edges(
    from = 1,
      to = 3) %>%
  trav_out_node(
    conditions = values > 7.0) %>%
  get_selection()

# Traverse from the edge `1`->`3`
# to the attached node where the edge
# is outgoing, this time filtering
# numeric values less than `7.0` for
# the `values` node attribute (the
# condition is not met so the original
# selection of edge `1`->`3` remains)
graph %>%
  select_edges(
    from = 1,
      to = 3) %>%
  trav_out_node(
    conditions = values < 7.0) %>%
  get_selection()

# Traverse from the edge `1`->`2`
# to node `2`, using multiple conditions
graph %>%
  select_edges(
    from = 1,
      to = 2) %>%
  trav_out_node(
    conditions =
      grepl(".*d$", label) |
      values < 6.0) %>%
  get_selection()
```

```
# Create another simple graph to demonstrate
# copying of edge attribute values to traversed
# nodes
graph <-
  create_graph() %>%
  add_node() %>%
  select_nodes() %>%
  add_n_nodes_ws(
    n = 2,
    direction = "from") %>%
  clear_selection() %>%
  select_nodes_by_id(nodes = 2) %>%
  set_node_attrs_ws(
    node_attr = value,
    value = 8) %>%
  clear_selection() %>%
  select_edges_by_edge_id(edges = 1) %>%
  set_edge_attrs_ws(
    edge_attr = value,
    value = 5) %>%
  clear_selection() %>%
  select_edges_by_edge_id(edges = 2) %>%
  set_edge_attrs_ws(
    edge_attr = value,
    value = 5) %>%
  clear_selection() %>%
  select_edges()

# Show the graph's internal edge data frame
graph %>%
  get_edge_df()

# Show the graph's internal node data frame
graph %>%
  get_node_df()

# Perform a traversal from the edges to
# the central node (`1`) while also applying
# the edge attribute `value` to the node (in
# this case summing the `value` of 5 from
# both edges before adding as a node attribute)
graph <-
  graph %>%
  trav_out_node(
    copy_attrs_from = value,
    agg = "sum")

# Show the graph's internal node data frame
# after this change
graph %>%
  get_node_df()
```

trav_out_until *Traverse outward node-by_node until stopping conditions are met*

Description

From a graph object of class dgr_graph, move along outward edges from one or more nodes present in a selection to other connected nodes, replacing the current nodes in the selection with those nodes traversed to until reaching nodes that satisfy one or more conditions.

Usage

```
trav_out_until(graph, conditions, max_steps = 30, exclude_unmatched = TRUE,
  add_to_selection = FALSE)
```

Arguments

graph	a graph object of class dgr_graph.
conditions	an option to use a stopping condition for the traversal. If the condition is met during the traversal (i.e., the node(s) traversed to match the condition), then those traversals will terminate at those nodes. Otherwise, traversals with continue and terminate when the number of steps provided in max_steps is reached.
max_steps	the maximum number of trav_out() steps (i.e., node-to-node traversals in the outward direction) to allow before stopping.
exclude_unmatched	if TRUE (the default value) then any nodes not satisfying the conditions provided in conditions that are in the ending selection are excluded.
add_to_selection	if TRUE then every node traversed will be part of the final selection of nodes. If FALSE (the default value) then only the nodes finally traversed to will be part of the final node selection.

Value

a graph object of class dgr_graph.

Examples

```
# Create a path graph and add
# values of 1 to 10 across the
# nodes from beginning to end;
# select the first path node
graph <-
  create_graph() %>%
  add_path(
    n = 10,
    node_data = node_data(
      value = 1:10)) %>%
```

```
  select_nodes_by_id(
    nodes = 1)

# Traverse outward, node-by-node
# until stopping at a node where
# the `value` attribute is 8
graph <-
  graph %>%
  trav_out_until(
    conditions =
      value == 8)

# Get the graph's node selection
graph %>%
  get_selection()

# Create two cycles in graph and
# add values of 1 to 6 to the
# first cycle, and values 7 to
# 12 in the second; select nodes
# `1` and `7`
graph <-
  create_graph() %>%
  add_cycle(
    n = 6,
    node_data = node_data(
      value = 1:6)) %>%
  add_cycle(
    n = 6,
    node_data = node_data(
      value = 7:12)) %>%
  select_nodes_by_id(
    nodes = c(1, 7))

# Traverse outward, node-by-node
# from `1` and `7` until stopping
# at the first nodes where the
# `value` attribute is 5, 6, or 15;
# specify that we should only
# keep the finally traversed to
# nodes that satisfy the conditions
graph <-
  graph %>%
  trav_out_until(
    conditions =
      value %in% c(5, 6, 9),
    exclude_unmatched = TRUE)

# Get the graph's node selection
graph %>%
  get_selection()
```

trav_reverse_edge *Traverse to any reverse edges*

Description

From an active selection of edges in a graph object of class dgr_graph, traverse to any available reverse edges between the nodes common to the selected edges. For instance, if an active selection has the edge 1->2 but there is also an (not selected) edge 2->1, then this function can either switch to the selection of 2->1, or, incorporate both those edges into the active selection of edges.

Usage

```
trav_reverse_edge(graph, add_to_selection = FALSE)
```

Arguments

graph	a graph object of class dgr_graph.
add_to_selection	an option to either add the reverse edges to the active selection of edges (TRUE) or switch the active selection entirely to those reverse edges (FALSE, the default case).

Value

a graph object of class dgr_graph.

Examples

```
# Create a node data frame (ndf)
ndf <-
  create_node_df(
    n = 4,
    type = "basic",
    label = TRUE)

# Create an edge data frame (edf)
edf <-
  create_edge_df(
    from = c(1, 4, 2, 3, 3),
    to =   c(4, 1, 3, 2, 1))

# Create a graph with the
# ndf and edf
graph <-
  create_graph(
    nodes_df = ndf,
    edges_df = edf)

# Explicitly select the edges
```

```
# `1`->`4` and `2`->`3`
graph <-
  graph %>%
  select_edges(
    from = 1,
      to = 4) %>%
  select_edges(
    from = 2,
      to = 3)

# Get the inital edge selection
graph %>%
  get_selection()

# Traverse to the reverse edges
# (edges `2`: `4`->`1` and
# `4`:`3`->`2`)
graph <-
  graph %>%
  trav_reverse_edge()

# Get the current selection of edges
graph %>%
  get_selection()
```

`trigger_graph_actions` *Trigger the execution of a series of graph actions*

Description

Execute the graph actions stored in the graph through the use of the `add_graph_action()` function. These actions will be invoked in order and any errors encountered will trigger a warning message and result in no change to the input graph. Normally, graph actions are automatically triggered at every transformation step but this function allows for the manual triggering of graph actions after setting them, for example.

Usage

```
trigger_graph_actions(graph)
```

Arguments

`graph`	a graph object of class `dgr_graph`.

Value

a graph object of class `dgr_graph`.

Examples

```
# Create a random graph using the
# `add_gnm_graph()` function
graph <-
  create_graph() %>%
  add_gnm_graph(
    n = 5,
    m = 10,
    set_seed = 23)

# Add a graph action that sets a node
# attr column with a function; this
# uses the `get_pagerank()` function
# to provide PageRank values in the
# `pagerank` column
graph <-
  graph %>%
  add_graph_action(
    fcn = "set_node_attr_w_fcn",
    node_attr_fcn = "get_pagerank",
    column_name = "pagerank",
    action_name = "get_pagerank")

# Add a second graph action (to be
# executed after the first one) that
# rescales values in the `pagerank`
# column between 0 and 1, and, puts
# these values in the `width` column
graph <-
  graph %>%
  add_graph_action(
    fcn = "rescale_node_attrs",
    node_attr_from = "pagerank",
    node_attr_to = "width",
    action_name = "pgrnk_to_width")

# Add a third and final graph action
# (to be executed last) that creates
# color values in the `fillcolor` column,
# based on the numeric values from the
# `width` column
graph <-
  graph %>%
  add_graph_action(
    fcn = "colorize_node_attrs",
    node_attr_from = "width",
    node_attr_to = "fillcolor",
    action_name = "pgrnk_fillcolor")

# View the graph actions for the graph
# object by using the `get_graph_actions()`
# function
```

```
graph %>%
  get_graph_actions()

# Manually trigger to invocation of
# the graph actions using the
# `trigger_graph_actions()` function
graph <-
  graph %>%
  trigger_graph_actions()

# Examine the graph's internal node
# data frame (ndf) to verify that
# the `pagerank`, `width`, and
# `fillcolor` columns are present
graph %>%
  get_node_df()
```

---

usd_exchange_rates *US Dollar exchange rates.*

Description

A dataset containing exchange rates from USD to all other currencies.

Usage

```
usd_exchange_rates
```

Format

A data frame with 196 rows and 3 variables:

from_currency the currency from which units will be used to buy units in the alternate currency (this is always USD)

to_currency the currency that is to be bought

cost_unit the cost per unit of the currency to be bought

---

visnetwork *Render graph with visNetwork*

Description

Render a graph object with the visNetwork R package.

Usage

```
visnetwork(graph)
```

Arguments

graph	a dgr_graph object, created using the create_graph function.

Examples

```
## Not run:
# Create a node data frame (ndf)
ndf <-
  create_node_df(
    n = 6,
    label = TRUE,
    fillcolor = c("lightgrey", "red", "orange",
                  "pink", "aqua", "yellow"),
    shape = "dot",
    size = c(20, 80, 40, 10, 30, 50),
    type = c("1", "1", "1", "2", "2", "2"))

# Create an edge data frame (edf)
edf <-
  create_edge_df(
    from = c(1, 2, 3, 4, 6, 5),
    to = c(4, 3, 1, 3, 1, 4),
    color = c("green", "green", "grey",
              "grey", "blue", "blue"),
    rel = "leading_to")

# Create a graph object
graph <-
  create_graph(
    nodes_df = ndf,
    edges_df = edf)

visnetwork(graph)

## End(Not run)
```

---

x11_hex *X11 colors and hexadecimal color values*

Description

Create a data frame containing information on X11 colors and their corresponding hexadecimal color values.

Usage

```
x11_hex()
```

%>% *The magrittr pipe*

Description

DiagrammeR uses the pipe function, %>% to turn function composition into a series of imperative statements.

Index